1991

Causal reasoning about distributed programs

William Samuel Lloyd
College of William & Mary - Arts & Sciences

Order Number 9212340

Causal reasoning about distributed programs

Lloyd, William Samuel, Ph.D.

The College of William and Mary, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

CAUSAL REASONING
ABOUT DISTRIBUTED PROGRAMS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

---

by

William Samuel Lloyd

1991

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

William Samuel Floyd

Author

Approved, July 1991

John P. Kearns

John P. Kearns, Dissertation Director

M. Ram

Ravi Mukkamala
Old Dominion University

William L. Bynum

William L. Bynum

Stefan Feyock

Stefan Feyock

John H. Drew

John H. Drew

DEDICATION

I dedicate this dissertation to my wife, Vicki.

Contents

ACKNOWLEDGEMENTS

I am grateful to Professor Phil Kearns for his intellectual, emotional and financial support.

List of Figures

ABSTRACT

We present an integrated approach to the specification, verification and testing of distributed programs. We show how the "global" properties defined by transition axiom specifications can be interpreted as definitions of causal relationships between process states. We explain why reasoning about causal rather than global relationships yields a clearer picture of distributed processing.

We present a proof system for showing the partial correctness of CSP programs. This system places strict restrictions on assertions. It admits no global assertions. A process annotation may reference only local state. *Glue* predicates relate pairs of process states at points of interprocess communication. No assertion may reference auxiliary variables; appropriate use of control predicates and vector clock values eliminates the need for them.

Our proof system emphasizes causality. We do not prove processes correct in isolation. We instead track causality as we write our annotations. When we come to a send or receive, we consider all the statements that could communicate with it, and use the semantics of CSP message passing to derive the appropriate postcondition.

We show that our CSP proof system is sound and relatively complete. We prove that we need to use only recursive assertions to prove that any program in our fragment of CSP is partially correct. Our proof system is, therefore, as powerful as other proof systems for CSP.

We extend our work to develop proof systems for asynchronous programming. For each of our proof systems, our motivation is to be able to write proofs that show that code satisfies its specification, while making only assertions that we can use to define the aspects of process state that we should trace during test runs, and check during postmortem analysis. We can trace the assertions that we make without having to modify program code or to impose additional synchronization or message passing.

Why, if we go to the trouble to verify correctness, do we want to test? We observe that a proof, like a program, is susceptible to error. By tracing and analyzing program state during testing, we can build our confidence that our proof is valid. We promote the view that verification is but one step, albeit an important one, in the task of developing software that meets its specification.

CAUSAL REASONING
ABOUT DISTRIBUTED PROGRAMS

Chapter 1

Introduction

1.1 Introduction

Though few programmers actually write axiomatic program proofs, most treat verification with awed respect. Told that some piece of software has been proven correct, they believe that it does what it is supposed to do. They treat a proof as an object of mystical faith.

It is true, of course, that program verification is a powerful tool. Verification can not only help programmers produce correct code, it can shape the way in which they think about their software throughout the development cycle.

Unfortunately, though, proofs, just like programs, can contain errors [GY76]. Proving correctness is hard, and verifiers sometimes make mistakes. What this means is that we must not see a program proof as a magical talisman that guarantees correctness. We must not think that our job is done when we have written a proof.

Instead, we must recognize that we need to validate the proofs we write. We must check out the assertions in our program annotations to see whether they accurately describe what happens during program execution. We must test to increase our confidence that our logic is sound.

Checking assertions is relatively straightforward for proofs of sequential programs. To

write an axiomatic proof, we construct a proof outline in which we associate each program statement with a precondition and a postcondition. These describe program state at the entry point to and the termination point of the statement. To validate the annotation, we can run standard white and black box program tests, checking to see whether the assertions correctly describe the state of the program as it executes.

There are many tools that we can use to test our assertions. Some sequential languages provide **assert** statements which halt execution if program state violates the condition defined in the statement. If they are not supported, we can insert print statements in the code to report on program state, write log records to a trace file for postmortem analysis, or use a debugger to set conditional breakpoints.

Whichever method we use, we rely on the fact that sequential program execution follows a single thread of control. The program passes through a totally ordered sequence of states. Before and after every statement executes, program state is directly accessible to us. We can easily determine, say, if $x = 3$ and $y = 6$ at the termination of statement γ in the program execution depicted in Figure 1.1.

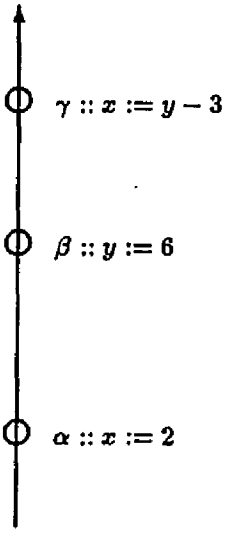

Figure 1.1: Time line for a test run of a sequential program.

But, suppose that the proof that we want to validate is of a distributed program.

In a distributed program, variable x may be defined in process π_1 and variable y in process π_2. If this is the case, it is no longer so easy to determine whether the assertion $(x = 3 \wedge y = 6)$ holds. Consider Figure 1.2. How do we determine whether $x = 3$ and $y = 6$ at the termination of statement γ_1 in process π_1? What does it mean to ask such a question?

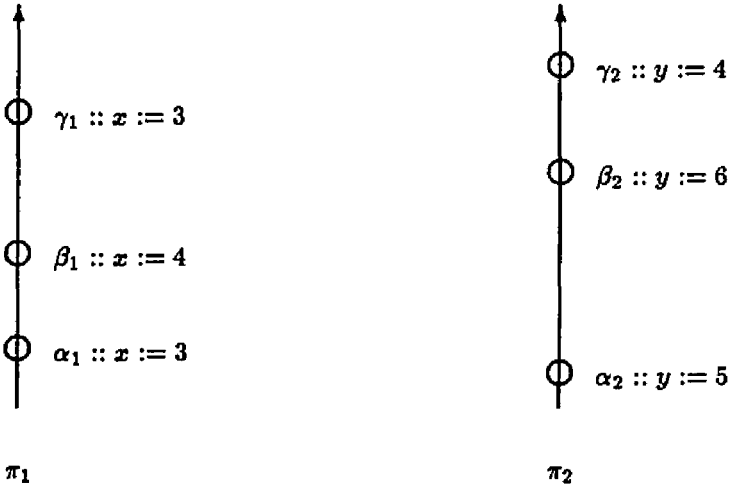

Figure 1.2: Time lines for a test run of a distributed program.

The answer is not intuitively obvious. We cannot just put an **assert** or a print statement in the code of π_1 or π_2, since only x is locally defined in π_1 and only y in π_2. We cannot insert a breakpoint which instantaneously suspends π_2 based on a condition in π_1, since each process in a distributed system runs asynchronously with the others.

The problem is that every process follows its own thread of control, with no shared memory and no global clock. Process states are only partially temporally ordered, not totally ordered like in a sequential program. To check a nonlocal assertion, we must *impose* some notion of global state on the process executions. We must, in general, add order to the system to be able to talk about what is simultaneously true in different processes.

Unfortunately, adding order order to impose global state is neither cheap nor harmless. Taking a distributed breakpoint or capturing a global snapshot requires additional message passing which may pervert the execution of the program. The global state which results

is consistent with some extension of the partial order of the execution, but is otherwise arbitrary. It may or may not detect a global condition in which we are interested.

We can capture all consistent global states only by tracing process states and doing postmortem analysis of the trace logs. We can then take every consistent cut or slice across process states to construct every global state. While tracing can also lessen the impact of testing on execution order, the cost of building all those global states renders it impractical for most programs. We face a combinatorial explosion if we try to make every consistent cut.

The cost of imposing global state, though clearly an important concern, is not the only issue the problem of checking nonlocal assertions raises. We commented that verification can influence the way we reason about our software. In this instance, it is in thinking about how we can validate assertions that a more fundamental question is exposed, that of the proper role of global state in reasoning about distributed programs. Global state is, after all, only an artifice. It plays no role in determining whether a program does its job correctly. Whether a program satisfies its specification or not depends entirely on the local states and interactions of its individual processes, not on some global state we build after the fact.

We believe, in fact, that thinking about global state distracts our attention from what is important in the execution of a distributed program. Rather than asking whether $x = 3$ and $y = 6$ at some instant in global time, we should ask what states of π_1 can affect the state of π_2. It is by answering questions like this, questions about the *causal* relationships between process states, that we best understand distributed processing. These relationships are also those which we need to specify in order to define the safety and liveness properties that our programs should exhibit, and those which we can effectively test as we validate our program annotations.

In this thesis, we build on these insights to define an integrated approach to specification, verification and testing which uses causal relationships between process states and

avoids use of global state. We specify the causal relationships which must hold between process states; we verify to show that these relationships hold in our implementation; and we test to increase our confidence that our proof correctly describes the path of causality as the program executes. Our methodology lets us demonstrate specification satisfaction by first asserting only what we can readily trace and analyze, and then tracing and analyzing precisely what we assert.

1.2 Integrating Specification, Testing and Verification

A roadblock in our path is that it often seems that the programming community is divided into two hostile camps: the testers and the verifiers. Those who test think that verification is just one more plaything for academicians, while those who write proofs imply that testing is merely a scam run by hackers too shiftless to be rigorous. Never, it appears, could the twain meet.

We believe that this mutual antagonism is misguided. Verification cannot replace testing, and testing can never eliminate the need for verification. Verification and testing should work together as closely related parts of a practical strategy for program development. The reasoning we use in verification can show us how and what to test, while the way we test can shape the kinds of assertions we make in our proofs. The energy and time we put into writing a proof can be recycled for use in testing, and what we learn from testing can shape our work in verification.

We want to tie verification to testing to encourage programmers to attempt correctness proofs, to tie tests to proofs to build confidence that verification did not go awry, and to make sure that both are firmly linked to specification. Testing is most helpful precisely in highlighting any mistakes we make in a proof. First we verify, then we test to see if we can find holes in our reasoning. A proof, barring error, shows the states a program will enter; a test increases our confidence that we reasoned about the program correctly. This

tactic makes testing the legitimate sibling of verification rather than its bastard cousin.

To make an integrated approach to specification, verification and testing more feasible, we believe it best to be able to think about our programs in the same way at each step. Though many programmers think in terms of program behavior as they specify, write, check and test their code, our approach focuses on states rather than on behaviors. We take this stance in part because we are dissatisfied with the behavioral methods we have seen, but more fundamentally, because we believe that reasoning about state is in general the best way to think about programs. Floyd [Flo67], Hoare [Hoa69], Owicki and Gries [OG76], Apt [AFDR80] and others have shown how code can be verified by proving assertions about program state. Lamport [Lam83a, Lam89] has written persuasively on the utility of specifying distributed program modules by defining permissible state transitions, and has defined what it means for an implementation of such a module specification to be correct. We want to adapt this type of reasoning to suit our needs, and to extend it a step further, so that we can have a consistent strategy for all three phases of program development.

Particularly with distributed software, execution tracing is a valuable technique for program testing. Tracing process state and performing postmortem analysis can minimize the effects of testing on an execution while capturing the aspects of state and the timing dependencies that we want to inspect. Our method is unique in that our efforts toward a consistent and efficient means of tracing motivate the way we use state to reason about distributed programs.

We view verification and tracing as mutually dependent. This being so, we cannot use the standard proof systems for distributed systems which make assertions about global state. Such assertions are difficult to check in process traces. Instead, we trace and make assertions about process state only. To make this possible, we use vector time in our predicates and our traces to track the flow of causality, and we avoid any notion of global program state. We gain as we test because testing is no longer done "by the seat of our

pants;" the proof defines the aspects of process state which we need to trace and tells how local states should be related. We gain as we verify because we never need to consider global state, which makes our proofs easier, and because we can reapply the work we put into writing the proof as we test.

Defined in this way, proof and test work hand-in-hand to see that a program meets its specification. Lamport's transition axiom specification method [Lam83a, Lam85, Lam89] works well as the first stage of our development methodology. A transition axiom specification defines a distributed program's safety and liveness properties. It encourages modularity, so complex specifications can be broken up into hierarchically organized parts whose implementations are simpler to verify and test. Though the formal underpinnings of the method make the correctness of its implementations verifiable, the actual specifications can be written in a way understandable to users and analysts uncomfortable with formal methods. The verifier uses this understandable specification and an axiomatic annotation of the implementation to show that a logical implication holds from code to specification. This implication establishes the correctness of the code, but because its validity depends on the validity of the proof, we still want to test to see that the states that the program enters are those the proof leads us to expect. Testing becomes a search for counterexamples to the supposed implication. The test, like the proof, focuses on the mapping between program states and the specification.

1.3 Outline of the Thesis

In the rest of Chapter 1, we review the literature which provides the background for this work. In Chapter 2, we discuss our approach to the development of correct distributed programs using axiomatic specification, program verification and proof validation with testing. We present a causal proof system for CSP programs in Chapter 3, and demonstrate that it is sound and relatively complete in Chapter 4. In Chapter 5, we adapt this proof

system for use with asynchronous message passing. Chapter 6 discusses execution tracing and postmortem trace analysis for the validation of proof annotations. Finally, Chapter 7 gives our conclusions and prospects for future research.

1.4 Background and Related Work

1.4.1 Overview

This thesis draws on the literature of several branches of computer science. Lamport's papers on specifying and proving distributed programs stimulated the attempt to develop a consistent approach to the entire software development cycle, though Lamport himself has shown little interest in testing. Readings on specification methods, program verification and temporal logic provided the necessary background for the work on proof systems for distributed programs. Study of the literature on testing, particularly on testing distributed systems, yielded many ideas, though often by means of negative example. Articles on vector time provided the means by which a verifier or tester could track the flow of causality.

1.4.2 Program Specification and Verification

Three of Lamport's papers define his approach to the specification of distributed programs. The first [Lam83a] describes the specification of modules in a concurrent program. It gives both an informal and a formal semantics of the specifications, touches on the method's roots in temporal logic, and defines state functions. It justifies the separate specification of safety and liveness properties, and provides a series of examples to show how the method is used in simple and in more complex situations. These same issues are reconsidered in the second paper [Lam85], which uses in its examples a difficult and somewhat obscure specification language.

The third of Lamport's papers [Lam89] tries to explain his ideas more clearly by

elucidating their underlying concepts. It describes how state transition definitions can be used in a specification, and shows how these definitions can be made by giving state functions and the rules which govern changes in their value. It stresses the distinction between interface and internal specification. Lamport makes it plain here that he is not proposing a particular specification language, but rather a general form for program specification, a foundation for valid and useful specification languages. He shows how specifications at different levels are related, and how valid code can be said to satisfy its specification. He also discusses the specification of concurrency and of nonatomic operations, and compares his method to some others.

To understand and build on these papers, we must understand the context of their development. Lamport's ideas grow out of the study of axiomatic program verification. Hoare's deductive method for program verification [Hoa69] is seminal for Lamport. Hoare proposed the use of axioms, assertions about program state, and rules of inference to prove correctness of sequential programs. He later applied these concepts to parallel programs [Hoa72]. Owicki and Gries [Owi75, OG76] furthered Hoare's efforts to enable proof of parallel program correctness in a shared memory environment. They gave a language which allowed concurrent, asynchronous modules, and showed how to demonstrate partial correctness, deadlock freedom and termination for programs in that language. They use proof outlines and process proofs, and must also do noninterference proofs since nonlocal assertions are a fundamental aspect of their proof system.

Owicki and Lamport extended this work in [OL82]. They distinguished between safety and liveness properties. (See [AS85] for a formal definition of safety and liveness.) Liveness properties in sequential programs are often shown by inductive arguments [BBFM82]. Lamport had previously considered their proof in multiple process programs [Lam77], introducing the *proof lattice*, a graphical technique for use in such proofs. Unfortunately, the method he suggested is hard to use. Owicki and Lamport improved it by using the proof lattice in conjunction with temporal logic to make rigorous, yet comprehensible

arguments about liveness properties. Their paper offers a brief introduction to temporal logic, defines proof lattices, reviews how temporal logic is used to express safety properties (as had been explained in [Kel76, Lam80a, Lam77, OG76]), and gives axioms for deriving liveness properties.

Levin and Gries [LG81] built on the work of Owicki and Gries to give a proof system for programs in a version of Hoare's CSP [Hoa78]. They also use process and noninterference proofs, and introduce the notion of satisfaction for parallel composition. Apt, Francez and De Roever [AFDR80] described a very similar proof system for CSP. Schlichting and Schneider [SS84] extended these ideas to give proof systems for other distributed programming paradigms such as synchronous models using remote procedure calls and asynchronous models using datagrams and virtual circuits. All these methodologies rely on nonlocal reasoning.

Owicki used an elegant technique to show the relative completeness and soundness of her proof rules for shared memory systems [Owi75, Owi76]. Apt built on her work in a series of papers which discuss what a proof system must be able to do to achieve relative completeness [ABM79, Apt81, Apt83].

Misra and Chandy take a different approach to proofs of CSP in [MC81]. While in the proof systems mentioned above individual process proofs rely on assumptions about the state of other processes, Misra and Chandy instead define an invariant which describes the sequence of communications of each process. These communication sequence invariants are external to the annotation of any process. Each process can be annotated in isolation, and then groups of processes combined using information about the communication sequence rather than about states of other processes, as in a satisfaction proof.

Soundararajan [Sou84] describes a similar methodology which restricts more completely than Misra and Chandy's even implicit nonlocal references. Each process is proved correct strictly in isolation, with no reference even to values in the communication sequence, and then a parallel composition rule uses the communication sequences to draw

nonlocal conclusions from process postconditions. Like Misra and Chandy, Soundararajan stresses isolation of process proofs. He ignores the state of communicating processes as he derives a process annotation, and so cannot give causality the central place it deserves.

One foundation of all this work on proof systems for parallel and distributed programs, and of Lamport's work on specification, lies in temporal logic. Temporal logic is a modal logic. Modal logics are concerned with necessity and possibility. The field of modal logic is reviewed briefly in [Par87b] and more completely in [HC68].

An introduction to temporal logic itself is Rescher and Urquhart [RU71], which clearly and thoroughly gives its history and a review of its branches and issues. Prior [Pri67] laid the groundwork for the modern study of temporal logic. Pnueli [Pnu81] then applied it to concurrent programs, as Burstall [Bur74] had done for sequential programs. Pnueli's work is extended by Manna and Pnueli in [MP81] and [MP88]. They describe the use of a linear time temporal logic for program specification, verification and development. Their logic is very expressive; it includes a next state operator and past (since) as well as future (until) operators.

Linear time temporal logics are not the only candidates for use with distributed systems. Branching time logics also have their proponents. These logics characterize a computation's execution traces as branching trees to highlight the concurrency and nondeterminism fundamental to parallel programming. The tree structure reflects the logic's view of time as only partially ordered. The partial ordering of time is taken in this view to mean that time is a group of distinct instants, with multiple real futures branching off from each instant. Since we have multiple futures, we can specify properties which hold only for certain execution sequences, not only those which are true for all sequences as with linear time logics. Emerson and Srinivasan give an overview of branching time logics from the point of view of computer science in [ES88].

Emerson and Halpern use a logic called CTL* (Computation Tree Logic) with both linear and branching time operators [EH86]. This logic is more expressive than either linear

or branching time logic alone. It includes both "path formulas" and "state formulas" to let it describe features of computations inexpressible in other logics. Clarke and Draghicescu [CD88] compare the expressiveness of CTL* with branching and linear time logics.

Pratt [Pra86] proposes a method for specifying processes which uses elements of temporal logic together with formal languages and partial orders. Processes in this model are sets of *pomsets.* A pomset, a partially ordered multiset, is a generalization of the string, since a string is a linearly ordered multiset; analogously, the process generalizes the view of a language as a set of strings. Operations on pomsets provide a powerful tool for specification of concurrent systems. Pomsets also offer a natural and realistic picture of concurrency: they model actual rather than interleaved concurrent events.

Lamport acknowledges that other models are more expressive than his, but maintains that this extra power is superfluous for specifying the safety and liveness properties of a distributed program [Lam80b, Lam83b]. If a specification is a contract between user and programmer, he says, it should describe only what the program must do, not what it can do or how it should do it. There is no need for a specification to describe properties which may hold in some execution sequences, or to define actual concurrencies among events.

In general, Lamport believes, it is best for a logic to include only the operators that we need. On this ground, he rules out the extra features of logics like that of Manna and Pnueli and of hybrid approaches like Pratt's. Past operators are not necessary for the description of liveness or safety, while the next operator and explicit descriptions of concurrency have no place in a specification of what properties a program is to have, but only perhaps in a description of an implementation.

Lamport maintains in [Lam80b] that branching time temporal logic, in addition to being too expressive in some ways, is deficient in one important way. He shows that the expression of liveness properties is not possible in the branching time logic he considers, and maintains that therefore only linear time logic is suitable for program specification. Emerson and Srinivasan [ES88], however, show that Lamport's analysis is correct only

for that particular branching time logic, and that others can express liveness and other similar properties.

Though Lamport may be wrong in this matter, his argument that the strengths of other logics are immaterial still has force. Lamport and Owicki emphasize in [LO81] that the goal of a logic for program specification and verification is not to be the most powerful logic, but to be able to express just what is needed in a useful, understand manner. The computer scientist is not a logician, so the aims of the two will not and should not be the same.

1.4.3 Vector Time

Vector time is defined in papers by Fidge [Fid88] and Mattern [Mat89]. Fidge gives vector clock maintenance algorithms for asynchronous and synchronous interprocess communications. He defines local event and state intervals, and discusses detection of synchronization errors using interval assertions. Mattern examines the causality relation between events in a distributed system, and introduces consistent cuts as event graph transformations which do not disturb a computation's causal relations. He describes an algorithm for updating vector clocks in an asynchronous system. He then discusses global state in terms of vector time, and defines concurrency as a relation between vector clock values.

Mattern shows that we can structure both cut sets and vector time as lattices. He draws an analogy between vector time and Minkowski's relativistic space-time, but notes that it can be carried only so far. Though attractive in that it seems to root vector time in physics, the analogy ultimately fails, because while two-dimensional relativistic time has a lattice structure, higher dimensional relativistic times do not.

Cheung [Che89] gives an algorithm for maintaining vector clocks in systems with synchronous message passing which improves on Fidge by allowing the same clock comparison to be made for synchronous and asynchronous systems. He also shows how vector time can be used to preserve precedence relations during process and event abstraction for

debugging.

Charron-Bost [CB89] extends Mattern's work to show how the number of consistent cuts determines a measure of the concurrency of a distributed computation. She discusses the interleaving model of concurrency, highlighting the difficulty of computing the number of linear extensions which carry a partial order into its equivalent set of total orders. Her analysis yields insight into the potential for characterizing a distributed computation by its consistent cuts, rather than by the linear extensions of its partial order, but also points out that the number of consistent cuts in a highly parallel execution can be very high.

1.4.4 Testing

Distributed testing has much in common with sequential program testing. Miller [Mil84] offers a concise review of sequential testing methods and tools. He defines levels of testing coverage which offer increasing degrees of testing thoroughness. Myers [Mye79] provides a more comprehensive, though always readable, study of sequential program testing. He contrasts black-box testing, in which the tester ignores the program's internal structure, with white-box testing, in which the test is designed in light of program internals. He discusses test-case design strategies for each, and looks at module and system testing. He also compares bottom-up and top-down approaches to incremental testing.

Garcia-Molina *et al.* [GMGK84] summarize the problems peculiar to distributed testing. They contrast the roles of bottom-up and top-down distributed debugging. The former is useful for finding errors in a single process, the latter for testing process interfaces. Garcia-Molina points out both the value of testing with program traces in capturing timing-dependencies, and the expense of recording them. He suggests tracing only significant events to reduce their cost, and considers which events are significant.

While this paper presents several valuable ideas, it treats testing too much as an art. There is no clear picture of the purpose of a test, of how we should relate test results and program specifications. Testing here is just an examination of system behavior under

some defined conditions. We find no real theory of testing, only a collection of strategies for comparing behaviors with rather ill-defined expectations.

Bates has tried to define more clearly the relationship between test behavior and program specification. His EBBA method [Bat88] constructs models of behavior by recording the occurrence of abstract, possibly composite, events. EBBA compares these abstractions with expected program behavior using syntactic pattern recognition. System actions are described with Bates and Wileden's Event Definition Language (EDL) [BW83]. If we also use EDL to specify system behavior, specification and test are firmly tied. Unfortunately, EDL describes arbitrarily complex events, not states, leaving its use for specification open to the criticisms we will make of behavioral specification methods.

Baiardi, de Francesco and Vaglini propose a somewhat similar model [BdFG86]. They describe a method for debugging programs written in ECSP, a language based on CSP for concurrent processing (though with extensions to allow more flexible interprocess communication) and on Pascal for sequential processing. Expressions which define events and/or program state specify program behavior. Specified and actual behavior are then compared by the debugger as the program executes. Like EBBA, this model offers a clearly defined relation between specification and test. However, it mixes behavioral and state-based specification, which makes either formal or informal program verification difficult.

The ECSP debugger relies on run-time testing. Since the debugger actively intervenes as the program executes, the computation's timing dependencies can be distorted. These *probe effects* [Gai86] are avoided only by careful action by the programmer, who must alter program execution to avoid them; message delivery, process synchronization and the occurrence of time outs can all be affected. Models which rely on tracing and post-mortem analysis, by contrast, may suffer from the probe effect to a degree determined by the amount of data they must log and the importance they place on maintaining concurrencies. To the extent that the overhead of logging and the importance of the analysis of concurrency can be reduced, the effect can be minimized.

Bugnet [Wit88], for example, reduces overhead by logging all messages but taking only periodic global checkpoints. It offers real-time replay of distributed programs to expose timing errors. To ensure that checkpoints are taken simultaneously, synchronized clocks are kept at each node. This constraint limits the generality of Bugnet, as does the fact that its logging increases message transmission time by a factor of three.

Igor [FC88] also allows program replay from traces. It uses existing virtual memory routines to take incremental checkpoints by periodically logging all dirty memory pages. Storage requirements can be controlled by having new checkpoints overwrite older ones.

Igor's performance depends on the paging behavior of the program being executed. There is no way to abstract relevant information or to filter unwanted data, and no guarantee that global precedences are maintained between processes. While the use of operating system code to support logging is interesting, Igor offers no general method for comparing execution and expectation; it is more an interesting experiment in logging than in testing.

Instant Replay [LMC87], like Bugnet and Igor, saves an execution log for program replay, but of shared memory systems. Storage overhead is reduced by logging only the order in which processes lock and unlock memory, and not the value of variables. Thus only the partial order of interactions is saved, with program reexecution necessary to get a more complete picture of an execution. While Instant Replay works only with lockable shared memory systems and is wedded to program replay, focusing on the partial causal order of an execution has been an influential idea in our methodology.

The literature on distributed testing stimulates our appetite for a consistent approach to specification satisfaction, but does not satisfy it. Wileden has commented [ML88] that most methods proposed for testing distributed software are engineering exercises rather than models of the process of testing. We need, he said, a theory of distributed testing. This thesis tries to answer that need. It attempts to show that, given a formal, yet practical, paradigm for specifying causal relationships, and a proof system which eschews

global state to reason about causality, we can join testing with verification to form not just "complementary methods for decreasing the likelihood of program failure [GG75]", but a consistent model for software quality assurance.

Chapter 2

Developing Correct Distributed Programs

2.1 Specifying Distributed Software

2.1.1 Axiomatic and Constructive Specification

A program specification is a contract between user and developer [Lam88a]. To be able to determine if the programmer has satisfied the contract, we must be able to prove logically that the software is correct. We must be able to reduce the problem of satisfaction to the proof of an assertion. The specification must be expressible as a mathematical formula. We can then prove that the software is correct by showing that the implementation logically implies the specification.

To show this implication, we make a mapping from the program onto the specification. The way we construct this mapping depends on the way in which we have written the specification of the program. Our specification can be either *constructive* or *axiomatic* [Lam83b]. A constructive approach prescribes the behavior of the program. It gives an abstract model of desired behavior; the specification is itself an abstract program. The programmer implements the abstraction by writing a less abstract, and probably more

complicated, program. An axiomatic specification, in contrast, defines the properties that the program must exhibit. The programmer implements it by writing code which manifests those properties.

Any constructive strategy must overcome two fundamental problems [Lam83b]. First, hierarchical decomposition is troublesome. The specification describes each behavior in terms of an abstract operation which, when implemented, may be executed as many program steps. To determine correctness, we must be able to define the sense in which the sequence of low level actions implements the higher level behavior. In a distributed system, other operations may execute concurrently with the actions in question. We must make sure that our sequence of operations is not affected by any concurrent actions, and so a general mapping from the implemented sequence to the abstract operation is not always easy.

Second, since the specification is itself a program, our ability to show the correctness of the implementation depends on the exactness of our understanding of what both specification and program do. In a distributed environment, even an apparently simple program can, by virtue of the interaction of its concurrent processes, behave in surprising ways. We may be left trying to project the behavior of the implementation onto a specification we do not really grasp.

If we assert exactly the properties that a program must have, as we do in an axiomatic specification, these problems do not arise. We do not define high level elementary operations and then try to project sequences of low level operations onto them. We define instead permissible sequences of states, where a state is an instantaneous "snapshot" of the program or of a constituent process by an omniscient observer. We merely specify how state should change, and show that our code is correct by proving that it effects the proper state transitions. We do not have to assume that we understand how a "simple" abstract program works and try to show that our implemented code does the same thing. We instead define as clearly and as precisely as possible what must characterize any correct

implementation, and then show that the actual implementation has those characteristics.

We must be careful in giving an axiomatic specification that we assert how state evolves and not what actions the program performs. It often seems more intuitive to talk about operations than about state transitions, to talk about a write to a file rather than about a change in the value of a buffer. However, the methods suggested by Hoare [Hoa69] and developed by many others (see [BBFM82] for a summary) for proving program correctness have shown the utility of making assertions about states in programs. Later work has extended these techniques for use with parallel and distributed software, for example [OG76, LG81, AFDR80, SS84]. These methods show how to make comprehensive and unambiguous assertions about program states, and can readily be adapted for use in program specification.

These assertions describe what a program is to do, but do not prescribe how it should be implemented. Even were we to give constructive, behavioral specifications whose semantics are clear (using, say, Milner's Calculus of Communicating Systems [Mil80]), programmers often see a behavioral specification as a prescription of how code is to be written, not just as a description of what the code is to do. This is unfortunate, because specifications should not dictate means, but should only define what must happen. A user can accurately define the changes which should occur in a data base given certain input, but it is the programmer's job to decide how best to store, access and update the data.

2.1.2 The Transition Axiom Method

Lamport has developed an assertional, state-based specification model which he calls the transition axiom method [Lam83a, Lam85, Lam89]. The transaction axiom method allows the formal specification of software systems. It can be used either for sequential or distributed systems, but it is especially useful for defining the latter. It does not claim to make formal specification easy, but it does offer clarity and precision without being too hard for use in real-life situations. It is suitable for use with formal verification of program

correctness.

Lamport's methodology defines all safety and liveness properties as allowed and required state transitions. It is applicable to both sequential and distributed software because it does not describe how a property is to be achieved, only that it must hold. We allow concurrency simply by including nondeterminism. This is consistent with our understanding that any program mapped onto a distributed system could be mapped onto a single processor with time-sharing [MP81], or even modeled by a Turing machine [Par87a]. Lamport believes that, while degree of concurrency is often important in the analysis of the performance of an implementation, it should play no part in its specification, which should just describe what the program must and must not do [Lam89].

We can interpret a transition axiom specification and its implementation as formulae in a common system of linear time temporal logic. We can therefore determine the logical relationship between them. If the implementation implies the specification, then it is correct. We can prove the correctness of any implementation by demonstrating such implication.

Transition axiom specifications support modularization. We can define modules, and prove their implementations correct, in isolation, and then combine them to form complex systems. At the system level, we need not worry about the internals of modules, but only about their interactions.

Also, the transition axiom method distinguishes between, and requires separate description of, safety and liveness properties. Safety properties state what a program may do, or, equivalently, what it is not to do. They ensure that something undesirable does not happen, that the program never assumes an unwanted state. Partial correctness and mutual exclusion are safety properties. Liveness properties are those which define what a program must do. They ensure that it enters some desirable state like termination or receiving service [OL82, Lam83a]. Since we use different techniques to prove safety and liveness properties [OL82], it is helpful to distinguish between them in a specification.

Lamport's specifications describe safety properties by defining a set of program states and a rule for state changes. The specification defines a *state function.* This function can take on certain values, the specification describing how these values may change. The transition axiom model uses the interleaving model of concurrency, in which the partial causal order of an execution is represented by its extension into all possible total orders. Therefore we understand that in execution the program passes through a sequence of states, and the state function maps from the set of states to the set of function values. The specification defines, for each state transition, the change of function value.

The mapping from states to function values may be many to one, so a transition may leave the function value unaltered. Thus an action of the system may involve many state transitions which do not change the value of the function. This allows hierarchical decomposition of the specification, since we can redefine the state function in terms of lower level program elements and show that changes in its value at the lower level can be mapped one-to-one onto those at the higher level.

As an example, consider a system for granting exclusive access to a resource. Users request the resource, use it and relinquish it. The system enters some set of states S. As it executes, it assumes some sequence $\sigma_0, \sigma_1, \sigma_2, \ldots$ of these states. The sequence depends both on the internal state of the system and on its interactions with its environment, namely the requests and releases it receives and the grants it issues.

We define a state function f which maps S to the set $\{I, II, III\}$. We specify for the state transitions $\sigma_i \rightarrow \sigma_{i+1}$ in an execution sequence the corresponding values of f, as shown in Figure 2.1.

We describe the state sequences which may occur by specifying what values the internal state of the system and its environmental interfaces may take on. The specification as given assumes that requests and grants are adequately defined, and for our purposes we do not need to give those definitions. We allow no interactions with the environment other than those given, so the specification completely defines the effects of requests, grants and

- $f(\sigma_0) = I$;
- $f(\sigma_i) = I$, $f(\sigma_{i+1}) = II$, and the transition is caused by a request;
- $f(\sigma_i) = II$, $f(\sigma_{i+1}) = II$, and the transition is caused by a request;
- $f(\sigma_i) = II$, $f(\sigma_{i+1}) = III$, and the transition is caused by a grant;
- $f(\sigma_i) = III$, $f(\sigma_{i+1}) = III$, and the transition is caused by a request;
- $f(\sigma_i) = III$, $f(\sigma_{i+1}) = II$, there is a pending request in σ_i and the transition is caused by a release;
- $f(\sigma_i) = III$, $f(\sigma_{i+1}) = I$, there is no pending request in σ_i and the transition is caused by a release;
- otherwise $f(\sigma_i) = f(\sigma_{i+1})$.

Figure 2.1: State transitions and changes in the value of f in the grant/release system.

releases.

Note that as system state changes, the value of f changes, but not every state change corresponds to a change in function value. When we implement the system, or specify it at a lower level, we may need to use many state transitions to effect, say, the allocation of a grant. This will not trouble us, because we have not specified each and every individual state transition. We have only specified that the state function f changes value just once during a grant. The details of how the grant is done do not concern us.

Our definition of f does not dictate the exact degree of concurrency an implementation must exhibit. In a distributed system, where events are only partially ordered [Lam78], no causal precedence can be determined for concurrent events. Because the transition axiom method uses the interleaving model, a specification allows concurrency where non-deterministic orderings are possible. For instance, if our system were implemented as a distributed system, the specification would *allow* (though not require) concurrent requests, but would not allow concurrent grants. Whichever of a set of concurrent requests is non-deterministically ordered first in an extension of the partial order, the effect is the same.

If no request is waiting for service, the "first" request changes the state from σ_i to σ_{i+1} and f's value from I to II, but any others, no matter how ordered, change the state but leave the function's value unchanged. Nondeterministic ordering of grants, on the other hand, is not possible, since we do not specify how the value of f changes if more than one grant is current. Our specification states that in any valid state, there must be at most one grant outstanding, so grants must be totally ordered, and therefore cannot be concurrent.

When we move to another level of specification or to an implementation, we will work in terms of functions whose values determine the system state. For instance, in a program these state functions reflect the value of its variables and its control state. If we can redefine f as a function of the state functions at this level, and show that every state transition at this level leaves the value of f unchanged or changes f as permitted in the original definition, then we show that the specification is satisfied.

The specification language we use will depend on the nature of the system and the level of abstraction we want. The transition axiom method is a specification method, not a specification language. A diagram representing state transitions might be appropriate for a simple system or a high level specification. For a lower level specification we might want to specify more explicitly the internal and interface state functions and the ways they change value. Using notation similar to that used by Lamport in [Lam83a], we could specify the safety properties of our request/grant system as shown in Figure 2.2.

Here $x \circ y$ means the concatenation of y onto x. The notation α[Allocate] in property 5 means that the changes indicated may occur only when a statement in module Allocate is executed. Conversely $\neg\alpha$[Allocate] in property 6 tells us that this change may occur only in the execution of some other module. This specifies the behavior of all other modules running with Allocate: none may change the value of Allocate's internal state functions or of *granted*, though the value of *request* may be changed by another module.

We say that the *control predicate* $at(A)$ is true if and only if control is at A's entry

MODULE Allocate
TYPE
pid: process id
pc : program counter value
INTERFACE STATE FUNCTIONS
$request$: *pid*
$granted$: *pid* or $NULL$
INTERNAL STATE FUNCTIONS
q : queue of *pid*
grantarg: *pid* or $NULL$
$INIT, REQ, GRANT, REL$: *pc*
SAFETY PROPERTIES
0) Initial conditions: $\mid q \mid = 0 \wedge grantarg = NULL \wedge in(INIT)$
1) $in(INIT) \supset \mid q \mid = 0 \wedge grantarg = NULL$
2) $at(REQ) \supset request \notin q \wedge request \neq grantarg$
3a) $at(GRANT) \supset grantarg = NULL$
b) $after(GRANT) \supset granted = grantarg$
4a) $at(REL) \supset grantarg \neq NULL$
b) $after(REL) \supset granted = grantarg$
5) changes α[Allocate] to $grantarg$,
q,
$request$:
a) $in(REQ) \rightarrow request_{new} = NULL \wedge q_{new} = q_{old} \circ request_{old}$
b) $in(GRANT) \wedge \mid q_{old} \mid > 0 \rightarrow$
$grantarg_{new} \neq NULL \wedge q_{old} = grantarg_{new} \circ q_{new}$
c) $in(REL) \rightarrow grantarg_{new} = NULL$
6) change $\neg\alpha$[Allocate] to $request$:
$\neg in(REQ) \wedge \neg in(GRANT) \wedge \neg in(REL) \rightarrow request_{new} = pid$

Figure 2.2: Lower level specification for the grant/release system.

point; $in(A)$ is true if and only if control is at A's entry point or inside A, but not at its exit point; and $after(A)$ is true if and only if control is at A's exit point [OL82].

Properties 1 through 4 are invariance properties, while property 5 states how the values of the state functions may change. Property 2, for instance, says that whenever control is at the entry point of REQ, then the value of *request* may not be in the queue and must not be equal to that of *granted*. Property 5a says that while control is at the entry point to or inside REQ, then *request*'s value is added to the queue and *request* is nulled out.

It is important for the development of our methodology that we understand what it means for such a lower level specification, or equivalently for an implementation of a transition axiom specification, to be correct.

A specification is formally a temporal logic formula of the form

$$\exists f_1, ..., f_n \, [X_1]$$

where the f_i are internal state functions and X_1 is a formula which defines the only allowed changes in the f_i and in the g_i, the interface state functions. We do not existentially quantify the latter because they must be specified at the implementation level even in the specification. Any system will exist in some environment, and its interfaces must be given according to that environment. For instance, when we specify a subroutine to be used in some programming environment, we must define its parameters according to the data types supported by that environment. For a discussion of this issue, see [Lam89].

X_1 defines the sequences of states which may occur by defining the values that the f_i and the g_i may assume on this state sequence. In the original specification of our example, f is the only internal state function, and X_1 is the description of how f can change value as internal state and environmental interfaces interact.

To show that a lower level specification (or an implementation) of such a specification is correct, we must show that it implies its specification. Our second specification uses internal state functions q, *grantarg*, $INIT$, REQ, $GRANT$ and REL, and interface state functions *request* and *granted*. It defines a formula

$$\exists \, q, \, grantarg, \, INIT, \, REQ, \, GRANT, \, REL \, [X_2],$$

where X_2 is a formula which states how the internal and external state functions may change value. We must prove

$$(\exists \, q, \, grantarg, \, INIT, \, REQ, \, GRANT, \, REL \, [X_2]) \supset (\exists f \, [X_1]).$$

We do this by defining the internal state functions of the original specification in terms of the internal state functions of the second specification, yielding X_1', and demonstrating $X_2 \supset X_1'$. (We do not have to redefine the interface state functions since they are specified at the implementation level in both specifications.)

We define a *state vector* for the second specification, a tuple of possible values for its internal state functions, and for the original specification a state vector consisting of a possible value of f. We then define a mapping, say F, from the former state vectors to the latter. Let the parameters of F be

1. a boolean which is true if the length of the queue is greater than 0 and false otherwise;
2. a boolean which is true if the value of *grantarg* is equal to some process id and false otherwise;
3. the value of the program counter.

We can define F as in Figure 2.3.

$$
\begin{aligned}
F(false, false, in(INIT)) &= I \\
F(false, false, in(REQ)) &= II \\
F(false, false, in(GRANT)) &= III \\
F(false, false, in(REL)) &= I \\
F(true, false, in(REQ)) &= II \\
F(true, false, in(GRANT)) &= III \\
F(true, false, in(REL)) &= II \\
F(true, true, in(REQ)) &= III \\
F(true, true, in(GRANT)) &= III \\
F(true, true, in(REL)) &= II \\
F(false, true, in(REQ)) &= III \\
F(false, true, in(GRANT)) &= III \\
F(false, true, in(REL)) &= I
\end{aligned}
$$

Figure 2.3: Mapping the lower level state functions to the values of f.

The specification tells us that in the initial state of an execution $F = I$. F changes value from I to II when the state transition is such that the value of the program counter changes value from $INIT$ or REL to REQ; from II to III when the program counter changes from REQ to $GRANT$; from III to I when the queue is empty and the program counter changes from $GRANT$ to REL; and from III to II when the queue is not empty and the program counter changes from $GRANT$ to REL. Other state transitions do not change F's value; for any two such states σ_i and σ_{i+1}, $F(\sigma) = F(\sigma_{i+1})$.

This pattern of changes in F's value corresponds exactly with the definition of f in the original specification. We can make a one-to-one mapping such that each change in F here projects onto a change in f. Therefore we can say that the second specification implies the first, and so is correct.

Likewise, in any correct implementation, we could again redefine f so that it changed only as allowed by the specification. This process of implication is valid because we can interpret both specification and implementation in terms of a common system of temporal logic. If there were no common logical system, as there often is not in constructive specification methods, the meaning of satisfaction would be unclear.

So far we have described only the specification of safety properties. To understand Lamport's treatment of liveness properties, we must develop a better understanding of temporal logic [Lam83a]. Temporal logic is the branch of modal logic in which the accessibility relation between possible states is the before/after temporal relation. Lamport defines a restricted logic suitable for program specification and verification. The execution of a program is seen in this logic as a sequence of state transitions. States and their transitions are defined loosely so that there be no requirement that a specified transition be implemented as one atomic operation. When we state that the variable x will change in value from 3 to 4, we do not want to specify whether this change be made in one step or as a series of operations. We want to specify just that the change take place. A state is therefore defined merely as a "freeze-frame" image of the program at some instant, a

transition as a change in state with no particular granularity required.

As we previously noted, Lamport represents the partial order of time in a distributed system by the interleaving model of concurrency. He proposes a linear time rather than a branching time temporal logic, so that at any time there is only one possible future. The possibility operator $\Diamond$ is therefore taken to mean that, for some predicate A, if $\Diamond A$ is true, then A is true now or at some instant in the one real future. Also, Lamport's logic contains no next operator $\bigcirc$ so that continuous as well as discrete time models can be accommodated. While these constraints limit the expressiveness of the logic, Lamport maintains that this is not a fault, since his goal is to develop not a completely general logic, but one which lets him specify and verify programs. What matters is that all safety and liveness properties can be expressed, so any additional power is unnecessary, and probably would lessen the clarity of specifications and program proofs [LO81].

We might ask why we need temporal logic at all. The problem with nonmodal logics is their weakness in describing the state transitions which occur during program execution [Lam83b]. A sequential program is a function from an initial state to a terminal state. We need only specify its input and output conditions, and need not mention the obvious temporal relation between them. When we specify a program which allows concurrency, however, we cannot consider only initial and final conditions. We must also consider the possible interactions between processes. Relative timing of statement executions among processes can produce different outcomes for code segments with equivalent input/output conditions. We must explicitly describe the temporal relations which should obtain if we are to specify adequately. Temporal logic gives us the power to do this. Lamport's restricted logic is designed to allow us to do just this and nothing more, and to do it clearly.

Since safety properties are those which are always true during the execution of a program, we could describe them without recourse to temporal logic. Liveness properties, on the other hand, require more involved temporal reasoning, and are best expressed

explicitly as axioms in temporal logic. A liveness property is specified in the transition axiom method as a liveness axiom, a relation between state functions which uses ordinary logic operators plus the temporal operators henceforth ($\Box$), eventually ($\Diamond$) and leads to ($\leadsto$), where $A \leadsto B \equiv \Box(A \supset \Diamond B)$.

For our detailed resource allocation specification, for example, we would define the liveness axioms as in Figure 2.4.

1. $in(REQ) \leadsto after(REQ)$;
2. $(in(GRANT) \wedge \mid q \mid > 0 \wedge grantarg = NULL) \leadsto after(GRANT)$;
3. $in(REL) \leadsto after(REL)$;
4. $after(REQ) \leadsto at(GRANT)$;
5. $after(GRANT) \leadsto at(REL)$;

Figure 2.4: Liveness axioms for the lower level specification.

The safety properties we defined specified what must be true at the entry to and on exit from REQ and what happens when control is inside it; our first liveness property tells us that if control is in REQ, then REQ must eventually terminate. Taken together, the properties form a complete definition of what a request entails. The other properties similarly together define what grant and release mean and how the states of the program are to be temporally related.

As with safety properties, we can define our liveness axioms in terms of the state functions at each level of specification and at the implementation level, and show that a lower level implies a higher one. For example, consider for our original specification the liveness axiom

$$(f = III) \leadsto (f = II) \vee (f = I).$$

Substituting the values of the parameters given above for $(F = I)$, $(F = II)$ and $(F = III)$, we can build a chain of relations to show the needed implication. Using the definition

of F, we have the equivalences shown in Figure 2.5.

$$\begin{aligned}(F \doteq III) &\equiv (grantarg = pid \wedge in(REQ)) \vee (in(GRANT)) \\ (F = II) &\equiv (grantarg = NULL \wedge in(REQ)) \vee (\mid q \mid > 0 \wedge in(REL)) \\ (F = I) &\equiv (\mid q \mid = 0 \wedge in(REL)) \vee (in(INIT))\end{aligned}$$

Figure 2.5: Equivalences between values of F and state function values.

Using both safety and liveness properties, we can then show the necessary implication as in Figure 2.6.

1. $(grantarg = pid \wedge in(REQ) \rightsquigarrow (grantarg = pid \wedge after(REQ))$
   $\rightsquigarrow (\mid q \mid > 0 \wedge grantarg = NULL \wedge at(GRANT)$,
   since $after(REQ) \supset \mid q \mid > 0$,
   $after(REQ) \rightsquigarrow at(GRANT)$, and
   $at(GRANT) \supset grantarg = NULL$.
2. But $(\mid q \mid > 0 \wedge grantarg = NULL \wedge at(GRANT) \rightsquigarrow$
   $(grantarg = pid \wedge after(GRANT)) \wedge (\mid q \mid \geq 0) \rightsquigarrow$
   $(grantarg = pid \wedge at(REL)) \wedge (\mid q \mid \geq 0) \rightsquigarrow$
   $((grantarg = pid \wedge at(REL) \wedge \mid q \mid > 0) \vee$
   $(grantarg = pid \wedge at(REL) \wedge \mid q \mid = 0)) \supset$
   $((in(REL) \wedge \mid q \mid > 0) \vee (grantarg = NULL \wedge in(REQ)) \vee$
   $(in(REL) \wedge \mid q \mid = 0)$.
3. This is equivalent to $(f = II) \vee (f = I)$.

Figure 2.6: Proof that the lower level specification guarantees $(f = III) \rightsquigarrow (f = II) \vee (f = I)$.

That we discuss temporal logic in the context of liveness properties does not mean that temporal logic has no place in specifying safety properties. We specify a safety property as an assertion that a state function must not change when it should not change, or equivalently, that it may change when some predicate is true. These assertions do not explicitly include temporal operators, but they may be interpreted as temporal logic formulas [Lam83a]. This allows us to interpret the entire specification as a set of formulas in temporal logic, and so to be able to prove program correctness. We do not always explicitly use temporal logic in the specifications because we want them not only to be

precise, but also to be understandable, and assertions of allowed changes are clearer to most programmers than are logic formulas.

2.2 Specification Satisfaction

2.2.1 Program Proofs and Program Testing

Once we have the specification of the properties a program should exhibit, we can begin the program's implementation. Unfortunately, software development is error prone. Programmers use many strategies to avoid bugs as they design and write their code. They may choose from structured design [YC79], structured programming [DDH72], desk checking [Wei71], team walkthroughs [You80], and high-level languages [Boo87, Pom84] among others.

While these techniques are helpful in reducing the number of bugs, they cannot entirely eliminate them [Gri81]. Motivating our work is the premise that specification satisfaction is most attainable if the software developer recognizes and can make use of the interdependency of specification, verification and testing.

Though testing is difficult and cannot demonstrate program correctness, it remains a valuable tool in producing satisfactory code [Woh85]. Program verification cannot take its place. Proving software correct is hard, especially with distributed software. While a proof may guarantee that a program is correct, the act of proving correctness is still a human process and is subject to error [DLP79]. Gerhart and Yelowitz cite errors in proofs of sequential programs in [GY76]. As for distributed software, Lamport, in giving a proof of a short code segment which implements the bakery algorithm, comments that several published proofs of that algorithm are in error [Lam87]. Trained computer scientists wrote the proofs and respected reviewers accepted them, but nevertheless the proofs were wrong. If this is so, how often must less skilled programmers in less optimal situations err in their efforts? We must recognize that no human attempt at verification can guarantee the

delivery of error-free software.

For any but the simplest programs, then, we cannot be sure that our annotation is valid. We take care as we make the assertions, but we must face the fact that we might make a mistake. If we take this as given, we can then accept the worth of testing. Good testing, though, is not easy, and testing distributed software is even more challenging than testing a sequential program. We may understand a distributed program as a set of interacting sequential programs [SK87]. All the bugs which crop up in a sequential program can still occur, as well as new types of errors. These added complications are due to the characteristics of a distributed system [GMGK84, LeL81]. A distributed program consists of a set of asynchronous processes, usually running on multiple processors, which communicate by message passing. There is no one thread of control, no common clock. There is no obvious way to assess the global state of the program, since our ability to find out what is happening at some instant is limited by communication delays and differences in local clock values. For example, a processor failure is immediately apparent in the execution of a sequential program, but a node failure in a distributed system may escape the attention of the user for some time. Moreover, since many errors are due to timing dependencies and improper process synchronization, bugs tend to appear and disappear and to be hard to reproduce.

In light of these difficulties and of those of verification, our best hope is to attack the problem of specification satisfaction on two fronts. We must take neither the hacker's attitude that verification is utterly impractical, nor the verifier's that testing is mere hacking. We must say instead that verification and testing can work together. We must define testing so that it is directed not towards program behavior, but, like verification, towards the relationship between program specification and process state.

2.2.2 Testing to Validate Proof Annotations

A Strategy for Tying Testing to Verification

When we determine whether a program satisfies its specification, we can write a proof of the program to exhibit its properties, so that we can make a mapping that demonstrates that the implementation implies the specification. We can then build confidence in the correctness of the proof by testing.

We test to see that what we asserted would be true of the states that the program enters, is actually true during our test runs. Using the axioms and proof rules which describe the semantics of the language constructs used in the code, we annotate the program to describe the states it will enter. We then increase our faith in the proof by analyzing trace logs of test executions to see if the assertions in fact hold during those executions.

Ideally, we would replace the program annotation predicates with writes to a log file. If we run the program with inputs chosen according to reliable black- and white-box test data selection strategies, and detect no errors when we analyze the traces, we can be more confident that the proof is, in fact, a proof. If we do detect the violation of an assertion, we will know that we need to rewrite the annotation (and even, perhaps, the specification), and then test it the same way. Through this iterative process we can hope to converge on a valid implementation of the specification.

Some trace strategies strictly limit *a priori* what we can trace, logging, for instance, only all modified memory pages [FC88] or only the partial order of interprocess communications [LMC87]. We believe that it is the verifier, while annotating the program, who can most effectively define what should be recorded. A program proof defines what states the program will enter in execution by asserting what values program variables will take on. If the point of a test is to build confidence in the validity of the proof, how better to do so than to determine whether the variable values called for by the proof are in fact

those assumed during the run?

We might object that this strategy corrupts the purpose of testing, that we test to see if the program meets its specification, not to compare it with a proof. A specification, though, is usually expressed in high-level terms. To test its satisfaction we must make a mapping between the states the program enters during a run and those the specification calls for. But this is just what a proof lets us do: it maps the actual variables and control structures used in the code back onto the specification [Lam89]. By tracing what the proof asserts, we can see if the relationship between code and specification has been properly defined, and so test whether the program meets its specification.

We might worry that letting the verifier choose what to trace will make tracing too expensive. We can avoid this penalty by using the principle of modularity as we specify, verify and test distributed programs. If our specifications are modular, then our module proofs, and so our traces, can be of reasonable size. If our specifications are not modular, then neither proofs nor tests will be manageable, and our approach would be impractical. We suspect, though, that this argument applies to any method for verifying and testing distributed software. Since even "simple" distributed programs can exhibit complex behavior, the developer of distributed software must think modularly, or face almost certain failure.

Obstacles to the Use of the Test Strategy

In the standard axiomatic proof systems for the verification of concurrent and distributed programs, for instance [OG76, AFDR80, LG81, SS84], two problems prevent the use of this simple proof validation scheme.

- **Auxiliary Variables:** The proof systems achieve completeness only through the use of *auxiliary variables* [Cli73, OG76] to encode the history and the control states of processes [Owi75, Apt81].

- **Non-Local Predicates:** The proof systems allow an assertion in the annotation of one process to refer to variables (possibly auxiliary variables) defined in another process.

Control predicates [Lam88b] can replace at least some auxiliary variables for the representation of process control state, but a control predicate is but an assertion about the program counter (PC) associated with a process. They are useful primarily useful in a process annotation when used to describe the control state of *other* processes. So, while control predicates may reduce the need for some auxiliary variables, they do nothing to restrict the use of non-local predicates.

Non-local predicates and auxiliary variables present no fundamental problem to the verifier as she writes her proof. Auxiliary variables are not part of the original program; they cannot affect the execution of the program, but only make formal reasoning about the program possible. *Non-interference proofs* [OG76] and *satisfaction proofs* [LG81] or *cooperation proofs* [AFDR80] justify global assertions. Non-interference proofs establish that every assertion, A, is invariant under the execution of every parallel statement, S. S is parallel to A if A appears in the annotation of one process and S is a in another process. Satisfaction and cooperation proofs tie process proofs together to account for the effects of interprocess communication on global assertions.

Global predicates and auxiliary variables do present a serious problem, however, when we try to trace program executions for postmortem analysis. Auxiliary variables cause a problem precisely because they are not part of the original program. We can alter the program to implement the auxiliary variables directly, but then we test a modified version of a program, not the original. Bugs in the altered program and timing perturbations introduced by our implementation of auxiliary variables make us question just what we are testing.

An even more fundamental problem lies in capturing and evaluating non-local predicates. Exactly how are we to log a predicate like $(x = 3 \wedge y = 6)$ when x and y are allocated on different nodes of a distributed system? Because the system is distributed, we cannot take an instantaneous snapshot of the system's global state. This difficulty points out an essential aspect of distributed processing: what it means for $\pi_i.x$ to be equal to 3 *and* $\pi_j.y$ to be equal to 6 is far from intuitively obvious. Global state in a distributed system is inevitably problematic.

2.2.3 Reasoning About Global State in Distributed Programs

Suppose, for example, that we are asked to test a simple CSP-like distributed program MUTEX. We want to see if MUTEX, with N processes $\pi_0, \ldots, \pi_{N-1}$, each like that shown in Figure 2.7, gives mutually exclusive critical section access. (As we will show in Section 3.7.1, MUTEX can also be seen as a correct implementation of the request/grant specification given in Figure 2.2.) The processes communicate synchronously to pass a

```
π_i:: do [] π_(i+N-1)mod N?token → if want_cs_i → critsec_i; want_cs_i := false
                                   [] not(want_cs_i) → skip
                                   fi
                                   π_(i+1)mod N!token
      [] true → do_other_i
   od;
```

Figure 2.7: Process π_i of the toy program MUTEX.

token around a logical ring. They may also exchange other messages in the critical section or in the noncritical code, but share no variables.

Were we testing a sequential program, our approach could be relatively straightforward. We might insert assert statements to stop execution between steps if some condition is not met, or we might trace the program's execution and do post-mortem analysis. In either case we could easily determine the program's state by checking the values of its variables

and program counter.

Testing MUTEX is not so easy. Interacting with it may pervert the course of its execution. Understanding its state requires an advanced degree in computer science. Knowing how to halt a run to take a breakpoint is worthy cause for a journal submission.

The cause of our trouble, of course, is that in the distributed world, unlike in the sequential, there is no one, global, view of time. We may speak of partially ordered virtual time in a distributed system, but it is clearer to say that events or process states are partially causally ordered by Lamport's "happened before" relation [Lam78]. Causality in a sequential program flows along in one stream, but in a distributed program, it breaks off in separate channels before coming together downstream, often in ways hard to predict. In MUTEX, the states in the execution of each π_i are totally ordered, but causality flows between π_i and $\pi_{(i+1) \bmod N}$ as the token is passed from the one to the other so that some, but not necessarily all, of the states of π_i are also causally related to some of those of $\pi_{(i+1) \bmod N}$.

An understandable reaction to this confusing state of affairs is to want to impose order on it. It may seem reasonable to construct global states for our distributed program, since we are used to thinking about program states from our experience with sequential programs. But because a distributed program consists of interacting sequential processes, we must define its global state as a function of process states and process interactions.

Several definitions of global state have been used in working with distributed programs. Lamport laid the groundwork with his definition of causal precedence and virtual time [Lam78].

Definition 2.1 Causal Precedence: *An event e causally precedes event e', denoted $e \rightarrow e'$, if and only if e and e' belong to the same process and e is executed before e'; e and e' belong to different processes, e is the transmission of a message and e' is the receipt of that message; or there is some event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.*

Definition 2.2 Virtual Time: *Any events which are not causally related are said to occur at the same virtual time, i.e., to be concurrent.*

Chandy and Lamport [CL85] used virtual time in describing an algorithm for recording *global snapshots* which yield consistent views of computations in strongly connected distributed systems. An initiating process records its local state. It next sends out a special message, the marker, on each of its outgoing channels. If a process receives a marker on some channel c and has not yet recorded its state, it records its state and records the state of c as empty. If a process has already recorded its state when it receives a marker on channel c, it records the state of c as the sequence of data messages it has received on c since recording its state. This algorithm records a view of process and channel states at one instant of virtual time, but at the cost, obviously, of (possibly extensive) additional message passing.

Miller and Choi [MC88] extended the global snapshot algorithm to implement breakpoints to make it more useful in testing. Their protocol uses additional message passing to ensure that every process halts at some one virtual instant.

Mattern [Mat89] used virtual time in defining *consistent cuts*, partitions of the events on execution traces of the system's processes, to talk about global state. A consistent cut consists of one event from each process such that no event in the cut is causally related to any other. Thus no message receipt is included if any event in the cut precedes the message's transmission. Any such partition defines a global state. Petri net theory and partial order semantics use the similar concept of *slices* [Rei88].

The interleaving model of concurrent executions [Lam80b], in contrast, extends the partial order given by an execution to all its possible total orders. This means that we can "single step" through totally ordered traces of the run and determine program state after each step. Each step in an execution defines a virtual instant.

In all these definitions, and necessarily in any definition of global state, we construct

our consistent view by *adding* order to the system. The action of a distributed program, even when we think of its goals in global terms, is strictly local. This is why causality is only partially ordered; it is determined only by the actions and intercommunications of the processes. To see the program globally, we must define some way to impose additional connections between the processes. We must send new messages, draw a cut line between unrelated events, or order previously unordered local states.

So long as we do not pervert the original causal order of the system, but only add to it, it might seem that constructing global state gives us new insight into the system's behavior, while doing no harm. Unfortunately, though, constructing global state is by no means harmless. Using global state to help us make sense of a program instead obscures our understanding of what it is doing.

At first blush, this seems odd. With most distributed programs, the properties in which we are interested appear to be global properties, like mutual exclusion in MUTEX. We are not particularly interested in the properties of individual processes, except as they contribute to our overall, system-wide goal. As Lamport pointed out in [Lam89], specifications should describe the safety and liveness properties a program should exhibit, not whether it must be implemented with sequential or distributed code. Similarly, when we test to see if the program meets its specification, what we want to know is if it does its job, not whether it is implemented as one or as sixteen processes.

Because we may specify that the properties in which we are interested are global, though, does not mean that we are wise to use global state when reasoning about program executions. Thinking about global state, in fact, directs the tester's attention away from specification satisfaction towards the details of implementation. Whether a distributed program exhibits the properties that it is supposed to have depends on the causal relationships between the local states of its processes, not on its global state.

Constructing global state cannot tell us about causality. Any valid global state is built up from concurrent process states, and concurrency means, by definition, that the local

states are causally unrelated. Concurrency and global state divert our attention from the causal relationships between local states that enforce the presence or absence of the properties we want to see. They encourage us instead to think explicitly about the system as a collection of distributed processes on which we must impose order.

That a specified property is "global," then, does not tell us that we should look for it by building global state. It tells us only that the property should be exhibited as the entire distributed program runs, not just as some particular process runs. First as we verify, and then as we test, we can restate a property expressed in global terms in terms of local states and their causal relationships, and so keep our thinking directed towards specification satisfaction.

The root of the problem with global state is the misleading analogy we draw between sequential and distributed programs when we decide to use global state to make the complexity of distributed processing manageable. In fact, we look at successive states in a sequential program not so much to capture separate states as to follow the path of causality as the program executes. When we move into the distributed realm, we shouldn't say that since we track the program states of sequential programs, we should construct the global states of our distributed programs. Instead, we should say that, just as we capture the flow of causality of sequential programs, we will track causality as our distributed programs execute.

Suppose, for instance, that we look at all global states in an execution of our program MUTEX and see that mutual exclusion holds in each. We should not be satisfied, because we have answered the wrong question. We need to ask not if mutual exclusion happens to hold, but rather if the program did what it had to do to ensure that it exhibited mutual exclusion. We answer this kind of question best by attending to causality, by analyzing the actions and interactions of processes, not by building global states. In our proofs, we should show causality as we map from process states and their relationships to the specifications. Then, as we test, we should look for those same local states and causal

relationships to see that our static analysis of the path of causality was correct.

Our objection to global state goes beyond that often leveled by partial order theorists. Disturbed by the emphasis on global state in interleaving semantics, they, like us, stress the causal ordering of system events and the absence of global time. Reisig, for instance, in [Rei88], writes that "...global views of [concurrent] systems are artifacts and should, if at all, be used with much care. The essential properties of distributed systems can certainly not depend on external views." He then goes on, however, to construct an elegant temporal logic in which "formulas will be interpreted over the slices" of system executions. A step in an execution, in this view, takes the system from a slice to a successor slice.

Were we to develop a test methodology based on this understanding of distributed executions, we would still be dealing with artificial global state. A slice or cut is as much an artifact as is an extension of the partial order to its total orders. Talking about causality in relation to global states adds a level of complexity to one's reasoning about distributed programs, but we believe that it does not give additional explanatory power. We avoid reference to global state, examining only the causal relationships between process states and never the concurrent relations we can construct between them.

We object to the use of global state not only because it can obscure our understanding, but also because it can be very expensive. If we do not construct every possible global state for an execution, we may run many tests and still not recognize even a regularly recurring fault. A distributed breakpoint, for example, stops all processes at the same virtual time, but it arbitrarily captures only one of the configurations of local states consistent with that virtual time. Even were we to single step through the executions of the processes, we would not, in general, see every valid global state, since we would still see only one extension of the partial causal order to a total order. And, as Miller and Choi note [MC88], using a breakpoint to detect a global condition which is the conjunction of causally unordered local predicates may be rendered impossible by the time delay imposed on the collection of data by the distribution of the system's processes. Our program might enter a state

which violates an assertion in our proof, and we would be unable to detect the violation.

We could avoid these problems with breakpoints by instead tracing each process and analyzing the traces after program termination. We could then check all concurrent states by making each possible consistent cut. We would then see any anomalous state which arises in a test run, but the computational cost may be high. Charron-Bost [CB89] has demonstrated that the number of consistent cuts in an execution equals the number of the antichains in the partially ordered set of its local states. In a highly concurrent system, this number approaches $\sigma!$, if σ is the number of local states. Analysis of this many states quickly becomes impractical as σ grows, so we would do better if we could avoid making global assertions.

2.2.4 Tracing Causal Relations

We can avoid these problems with global reasoning if we let our efforts toward tracing motivate the way we reason about distributed programs. Local traces can record all causal relationships, without imposing an arbitrary total order as a global trace must. Individual process traces can easily maintain the original partial ordering, and so record a run accurately.

Collecting local traces and checking causal relationships can be much cheaper tracking global state. Analyzing causal relations in a distributed system means comparing states in one process, if a local operation is performed, or in a pair of processes, if an interprocess communication occurs. Even if we had to take a brute force approach and compare every pair of states in the process logs, we would need to make only on the order of σ^2 comparisons, where, as before, σ is the number of local states. With application specific knowledge, we can usually reduce this number significantly.

We can use our toy program MUTEX to illustrate these points. If we used breakpoints, we might run multiple tests without finding anything wrong even if mutual exclusion failed on every run, because we might not be lucky enough to halt in the state in which it failed.

We would do better to build all global states from process state traces. With MUTEX, we could determine π_i's state solely by its control state, since we aren't interested in the value of its variables. We can define the control state of a process by the value of its program counter and by its position in the causal order of process states.

We can efficiently represent the latter by using the vector clocks of Mattern [Mat89] and Fidge [Fid88]. Vector time accurately and completely records the partial order of significant events and states in a distributed system. Its usefulness in testing and debugging has been demonstrated by Fidge [Fid88], Cheung [Che89] and Helmbold [HMW89]. While Lamport's logical clocks [Lam78] guarantee that if an event e with logical timestamp T "happens before" event e' with timestamp T', then $T < T'$, vector clocks give the stronger guarantee for significant events e and e' that $T < T'$ if and only if $e \rightarrow e'$. We can define *significant* to suit the needs of the application, but at a minimum we must consider changes in state caused by sends and receives significant, if vector time is to represent the system's partial temporal order.

The trace system can maintain vector clock values as the program executes at little cost, and record them, along with statement labels to represent program counter values, to represent control state in the N process trace logs. We assign each process its own vector clock. The clock of process π_i is a vector $T_i = [T_{i,0}, T_{i,1}, \ldots T_{i,N-1}]$ of integer values, where N is the number of processes. At program initiation, each vector clock is set to $\vec{0}$. Significant local state transitions update the clock, using only locally available information.

When a significant local state change occurs in process π_i, π_i increments element $T_{i,i}$. During a synchronous communication between π_i and process π_j, π_i increments element $T_{i,i}$ in its vector, π_j increments $T_{j,j}$, and π_i and π_j exchange vectors, each setting each element of its vector to the maximum of the value of that element from its vector and the corresponding element of the other vector [Che89]. At the exit point of a synchronous

communication, then,

$$T_i = T_j = synch(i, j, T_i^{old}, T_j^{old})$$
$$= \sup([T_{i,0}, \ldots, T_{i,i} + d, \ldots, T_{i,N-1}], [T_{j,0}, \ldots, T_{j,j} + d, \ldots, T_{j,N-1}]),$$

where $d > 0$. [1]

For instance, suppose MUTEX had 3 constituent processes. If P_0's vector clock $T_0 = [4,4,0]$ and P_1's clock $T_1 = [3,5,3]$, then if P_0 and P_1 communicate, the clock values at the completion of the communication will be $T_0 = T_1 = [5,6,3]$.

If we were to think in global terms, we might say that MUTEX gives mutually exclusive access if $\bigwedge_{i=1:n}[in(critsec_i) \supset \forall j \neq i : not(in(critsec_j))]$, where $in(critsec_i)$ means that control in π_i is at the entry point to or inside its critical section. (Lamport, for example, uses a similar assertion in an example in [Lam88b].) Since this means that we want to know the states of all processes when any given process is in its critical section, we can note the vector time associated with a critical section and build all global states which include the critical section. It is simple to tell which process states are concurrent by checking that their associated vector clock values are incomparable, that is, for π_i and π_j, that $T_{j,i} \not> T_{i,i}$ and $T_{i,j} \not> T_{j,j}$. Unfortunately, though we may find it easy to build all interesting global states, there may be many of them for each critical section execution, so our globally-oriented test could still be very computationally expensive.

Suppose, for example, that we had a four-process MUTEX program. Figure 2.8 shows as the shaded region on the four process time lines the states in π_1, π_2 and π_3 which are parallel to the state labeled $\mathbf{in(cs_0)}$ on the time line of π_0. If, as shown, π_1 has 2 states, π_2 has 3 states and π_3 has 2 states in the shaded region, then there are $2(3)(2) = 12$ global states which include π_1's critical section execution. With more processes or more states, the number of global states would grow rapidly.

If we think in terms of causality, we can do the job more cheaply and more thoroughly.

[1] In practice, it is usual for $d = 1$, but we allow d to take on values greater than 1 if the application demands it, and to achieve formal completeness in proof systems which rely on vector time.

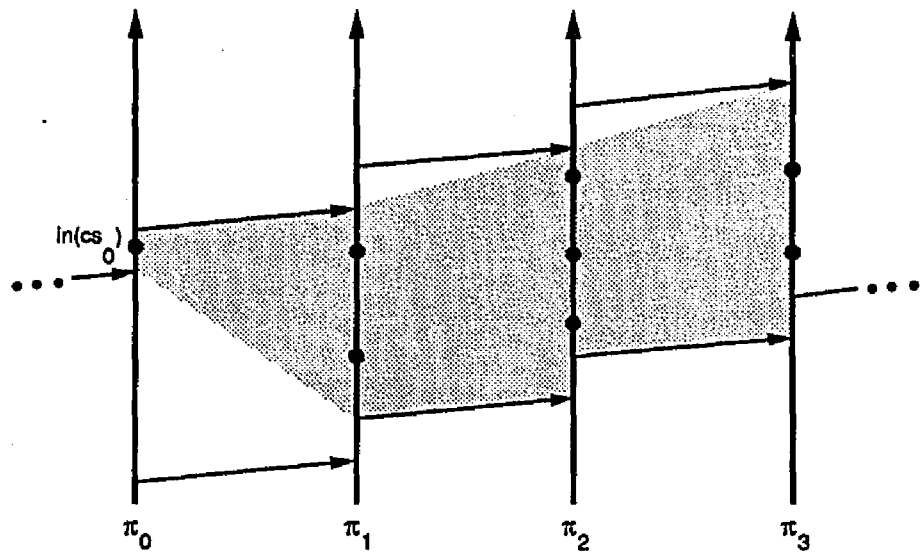

Figure 2.8: Parallel states of a critical section execution in MUTEX.

In writing our proof, we could restate the global mutual exclusion property specification in terms of local states: for any two critical section executions *critsec* and *critsec'*, either *in(critsec)→in(critsec')* or *in(critsec')→in(critsec)*. This means that when we analyze the trace logs we don't need to group each critical section record with all combinations of records of concurrent process states, or even to compare it with all concurrent records. We can just determine whether all pairs of critical section executions in a run are causally related, which is to say, whether the critical sections are totally ordered. All we need to do is to sort the vector timestamps of the critical section trace records, which we can do in $O(m \log m)$ time, if m is the number of critical sections. If any two records in the sort are not ordered, we know that mutual exclusion did not hold, and we can go to the traces to see what happened for debugging. As we then follow the causal connections through the traces, we will be able to see clearly the precedence relations which enforced or failed to enforce mutual exclusion.

2.2.5 Satisfying Transition Axiom Specifications

The question remains whether process-oriented, causal reasoning of this kind is adequate for demonstrating that an implementation satisfies an arbitrary property given in some general specification methodology like the transition axiom method. That is, can we show the existence of all the properties in which we are interested with local, causal reasoning? In chapter 4 we give a formal proof of the relative completeness of a proof system which relies only on causal reasoning, but we can also give here a more intuitive explanation of why global reasoning is not necessary.

To show that a transition axiom specification is satisfied, we must reinterpret it in terms of state functions of its implementation. These state functions are functions of the data and control state of the processes. If we can fully describe the how the process state functions change value in terms of local actions and interactions, we can make the necessary mapping to show satisfaction.

But the state transitions of a process in a distributed system can only be caused by what occurs locally in that process and by its communications with its environment. A process affects another only by communicating with it, and in any distributed system interprocess communications are effected explicitly via communication commands. Processes which do not communicate are not causally related; in determining the state of a given process, worrying about the impact of the state of an isolated process is pointless. The state of another process is immaterial, except at points at which that process communicates with ours.

This coincides nicely with what transition axiom specifications specify. A transition axiom specification defines deterministic relationships. Concurrency enters only as nondeterminism, that is, only by what is not explicitly specified. The specification describes permissible and required *sequences* of states. States in a sequence are ordered, and thus are causally related, so since we can describe all causal relationships in local terms, we

can show the necessary implication from implementation to specification.

Chapter 3

Proof Systems for CSP

3.1 Integrating Proofs Into the Development Methodology

The missing link so far in our software development methodology is a system for annotating distributed code. We know that we want to specify desired state transitions using the transition axiom method, and that we want to test to build confidence that the mapping we make from code to specification is valid. We need now to define the axioms and proof rules we will use in our proofs.

We need formal proofs to be able to define state functions for our implementations; for all but the simplest code, we cannot expect to be able to define them correctly without annotating the program. In this chapter we will describe a proof system for CSP, Hoare's language for *C*ommunicating *S*equential *P*rocesses [Hoa78]. Hoare designed CSP for the description of algorithms, and it has become one of the most commonly used vehicles for discussion of theoretical aspects of distributed processing. Our proof system is for proofs of partial correctness. Though our partial correctness proofs would be useful as the basis for liveness proofs, we will discuss proofs and tests of liveness properties only briefly and informally in discussion of several examples.

First, we will describe CSP and the subset of it we will use. Next, we will briefly discuss three well-known systems for proofs of partial correctness, and show that their use of auxiliary variables and non-local assertions is incompatible with our strategy. We will then describe axioms and rules that allow strictly causal proofs, and give examples of their use. In the following chapter we will show that our proof system is sound and relatively complete.

3.2 Informal Description of CSP

CSP describes distributed processing in which each process, $\pi_1, \pi_2, \ldots, \pi_n$, has its own memory and communicates with other processes only through message passing. All message passing is synchronous: a sending or receiving process blocks until the process with which it wants to communicate reaches its corresponding communication command. A process may be ready, in which case its execution can proceed, blocked for a communication, or terminated.

The commands defined in CSP are:

assignment :	$x := e$
skip :	*skip*
send :	$\pi_j!e$
receive :	$\pi_j?x$
sequence :	$S_i^1; \ldots; S_i^m$
parallel :	$[\pi_1 :: S_1 \parallel \ldots \parallel \pi_n :: S_n]$
alternation :	**if** $[]_{j=1:m} b_j \rightarrow S_i^j$ **fi**
repetition :	**do** $[]_{j=1:m} b_j \rightarrow S_i^j$ **od**

A process whose next statement is an assignment or *skip* is ready. An assignment in process π_i assigns the value of expression e to variable x defined in π_i. A *skip* has no effect.

Communication from π_i to π_j is effected by the send command $\pi_j!e$ and the receive $\pi_i?x$. A communicating send and receive make each process wait for the other to be ready to execute its communication command, and then execute a distributed assignment. Communicating processes *synchronize* at the send/receive pair. Variable x defined in π_j receives the value of expression e, as e is evaluated in π_i. In a complete version of CSP, send and receive commands allow assignment of a list of expressions to a list of variables, and include a template which types each expression or variable, but we will ignore these features.

The sequence command allows sequential program composition. $S_i^1; \ldots; S_i^m$ are statements in π_i, as indicated by the subscripted i, and are executed in the sequence given by the superscripts $1, 2, \ldots, m$, as the command's name implies. The parallel command executes the processes $\pi_1, \ldots, \pi_n$ in parallel. Process π_i executes statement S_i. The parallel command terminates when every component process has terminated. No data passes from the process which executes the command to the constituent processes, or from the constituents to the process which issues the command.

The alternation and repetition commands are made up from guarded commands. The notation

$$[]_{j=1:m} b_j \rightarrow S_i^j$$

abbreviates

$$b_1 \rightarrow S_i^1 \; [] \ldots [] b_n \rightarrow S_i^n.$$

Here b_k is the kth boolean guard in the command, while S_i^k denotes the statement (in π_i) guarded by that boolean. Our subset of CSP does not include the input–output guarded commands, in which the guarded command has the form

$$[]_{j=1:m} b_j; c_j \rightarrow S_i^j,$$

where c_j is a send, receive or skip statement. For partial correctness we can consider this

form equivalent to the guarded command

$$[]_{j=1:m} b_j \rightarrow c_j; S_i^j.$$

The second form leaves the program more exposed to deadlock, since the guard may be true when the communication command is blocked. Since we are not discussing liveness in this chapter, this distinction does not concern us here. We say that a guard fails if b is false. It is ready if b is true and c is *skip*. It is prepared to communicate with process π_j if b is true and c_j is $\pi_j!e$ or $\pi_j?x$, and is blocked until sender and receiver synchronize, when it becomes ready.

An alternation command nondeterministically selects a guarded command whose guard is ready, and executes $c; S_i^j$. If all guards fail, the process aborts. A repetition command repeats the nondeterministic selection of ready guarded commands until every guard fails, and then terminates.

The original definition of CSP allowed only receive commands in guards. We, like Levin and Gries [LG81] and Schlichting and Schneider [SS84], allow output guards. Also, our version of CSP, like those just mentioned and that of Apt [Apt83], does not allow distributed termination, which is a feature of the original definition. Distributed termination lets a loop in one process terminate because another process terminates; a guard with a communication command fails if the process given in the communication command has terminated. To simplify the semantics of our version of CSP, we require that explicit termination messages be sent so that booleans can be set to terminate a loop.

We assume that in the absence of deadlock, the execution of a ready process will progress in some bounded and finite period of time. This is our only assumption about scheduling.

3.3 Untraceable Proofs of Distributed Programs

The proof systems for CSP of Apt, Francez and De Roever [AFDR80], Levin and Gries [LG81] and Schlichting and Schneider [SS84] give us the power to show partial correctness, but at the cost of reliance on global thinking and auxiliary variables.

Though there are some significant differences between these methodologies, they are essentially similar. Each allows a verifier to derive an invariant which describes the states through which a given CSP program will pass. The invariant describes, for each statement in the program, the state of the program before and after the statement executes. The verifier uses axioms and proof rules which define the semantics of CSP commands to draw conclusions about program states and so construct the invariant. To simplify its derivation and make it easier to use, the verifier builds the invariant in the form of proof outlines, as suggested by Owicki and Gries [OG76]. In a proof outline, each statement is bracketed by assertions, called the pre– and the post–conditions, which describe program state at entry to and on exit from the statement. As Owicki has shown [Owi75], annotations in this form let us reason about the relatively simple pre- and post-assertions rather than about the one more complicated invariant assertion which describes the entire parallel program. If, in a proof outline annotation, p is the pre–condition and q the post–condition of a statement S, we can write $\{p\}S\{q\}$, which means that if p is true at the entry point of S, and if S terminates, then q will hold at the exit point of S.

In each of these proof systems, we show that a CSP program is partially correct by proving that each process is correct in the absence of message passing, and then showing that we can combine these process proofs to make a program proof if the effects of their synchronous communications are considered. Each system allows us to make any assertion which is valid in it as the postcondition of an input or output command in a process annotation. In isolation, these assertions seem "miraculous", since a communication command cannot terminate in isolation. (This use of miraculous postconditions in a process

annotation is justifiable formally because if a statement S cannot terminate, then any post–assertion q is vacuously true.) Obviously, though, we would not want to make just any arbitrary assertion. Instead, when we combine the isolated process proofs, we must be able to demonstrate that all post–conditions of communication commands follow from the semantics of synchronous communication.

In the Levin-Gries and Schlichting-Schneider systems, we combine process proofs with a *satisfaction* proof. In showing satisfaction, we prove for each matching pair of send and receive commands $\pi_i!y$ and $\pi_j?x$, that if in isolation we showed $\{p\}\pi_i!y\{r\}$ and $\{q\}\pi_j?x\{s\}$, then

$$(p \wedge q) \supset (r \wedge s)^x_y,$$

where the notation p^x_y indicates the substitution of y for each free occurrence of x in p. The satisfaction proof must show that all global references are justified by the assignment of y to x.

Levin and Gries forbid reference to the program variables of other processes in a process annotation. Their assertions may reference shared auxiliary variables as well as local variables. The verifier adds the auxiliary variables as necessary to the code of a process to record its control and data state and the history of its computation. The auxiliary variables do not alter the course of execution of the process, but only allow the verifier to define its state completely. The verifier then uses them in assertions about that and other processes to relate process states and so build global state.

Levin and Gries use the shared auxiliary variables to match sends and receives which could communicate in some execution, and to rule out those which could never communicate. They modify the semantics of the CSP communications commands to allow auxiliary variables to be updated and transmitted as part of a send or receive. Since it allows references to the auxiliary variables of other processes in the annotation of a process, their proof system requires noninterference proofs like those used in the Owicki-Gries proof sys-

tem for shared memory programs [OG76], except that noninterference must also be shown for statements parallel to matching send/receive pairs.

Schlichting and Schneider differ from Levin and Gries in that they allow the program variables of any process to appear in a process annotation. This lets the verifier relate the states of different processes using program variables as well as auxiliary variables, but does not eliminate the need for auxiliary variables to represent process control state and history.

In the proof system of Apt *et al.*, the annotation of each process references only local state, so no noninterference proof is needed. It uses locally defined auxiliary variables in process annotations to record local state. Then, *global invariants* use the auxiliary variables of different processes to describe global properties which must hold throughout a program's execution.

A global invariant relates process states so that communicating sends and receives can be matched and their miraculous postconditions resolved in a *cooperation* proof. But the global invariant is held to be true only outside arbitrary bracketed sections of the code. Inside these bracketed regions auxiliary variables are updated and a communication command is performed. The bracketed statements in effect constitute one atomic statement, so that the auxiliary variables can be updated "at the same time" that the communication occurs. This accomplishes what Levin and Gries achieved with their extension to the semantics of the communication commands. In the Apt system, if $S_!$ and $S_?$ are bracketed sections of communication commands and assignments to auxiliary variables such that the communication commands match, the cooperation proof shows that

$$\{pre(S_!) \wedge pre(S_?) \wedge GI\} S_! \| S_? \{post(S_!) \wedge post(S_?) \wedge GI\}$$

holds, where GI is the global invariant.

3.4 An Example, and the Problem It Reveals

To see the problems that the use of auxiliary variables and non-local state in these proof systems entails, consider the CSP program Minset and its annotation, adapted from Levin and Gries [LG81]. Program Minset defines processes A and B, where A has a set of integers $\mathcal{A} = \{a_i | i \in 1 : N\}$ and must send the minimum of the set to B. A executes, in parallel, N processes Least(i), $i \in 1 : N$, as shown in Figure 3.1.

```
[A || B]

A:: [||_{i=1:N} Least(i)]

Least(i)::
  integer mymin, theirmin, mysize, theirsize;
  mymin, mysize := a_i, 1;

  do
  []_{j=1:N∧i≠j} 0<mysize<N;Least(j)!(mymin,mysize) → mysize := 0

  []_{k=1:N∧i≠k} 0<mysize<N;Least(k)?(theirmin,theirsize)→mymin,mysize:=min(mymin,theirmin),
                                                          mysize + theirsize
  od;

  if mysize=0 → skip
  [] mysize=N → B!mymin
  fi

B::
  if []_{i=1:N} Least(i)?m → skip fi
```

Figure 3.1: Program Minset

A assigns each Least(i) a value a_i from $\mathcal{A}$. Least(i) maintains mymin, the minimum of any values it has received from other Least processes and its initial a_i, and mysize, the count of all the values about which it knows, including a_i. Least(i) nondeterministically chooses either to send its mymin and mysize to any other Least process willing to receive them, or to receive from some other Least its minimum and size. If Least(i) sends its values to Least(j), its job is done and it terminates. If it receives values from another, it

must continue until it has sent out its values or has directly or indirectly accounted for all N values in the original set and can send B the minimum of the entire set and terminate. B blocks until it receives the minimum value and then it terminates.

Figure 3.2 shows the Levin-Gries proof annotation for Minset. The annotation introduces, for each Least(i), the auxiliary variable set_i, which contains the indices of the values from A's set for which Least(i) knows the minimum. The minimum and cardinality of set_i are mymin and mysize. Likewise, B maintains set_0, the set of values it knows about. The predicate UNION is a global invariant, true everywhere.

We refer the reader to [LG81] for Levin and Gries' proof of Minset. Our interest lies in considering how we would trace an execution of Minset to look for errors in the annotation, since our plan is to trace what we have asserted. The problem is that the program proved is not the original, but the original augmented by auxiliary variables. To trace its execution, we would have to modify Least(i)'s code to include and update set_i. Worse, the augmented code of Least(i) refers to set_j, Least(j)'s auxiliary variable, and B's code refers to set_i as well as to its own set_0. We could define B's set_i as a local replica of Least(i)'s set, but to send Least(j) $set_i \cup set_j$ would require either an extra message from Least(j) to Least(i) or an extra assignment in Least(j).

This is not a problem peculiar to the proof of Minset. The Levin-Gries method in general requires the use of shared auxiliary variables, while the Schlichting-Schneider system lets remotely defined program variables appear in a process annotation. We might be able to resolve such problems by maintaining local copies of all nonlocal variables and updating them appropriately, adding communication as necessary, but this would be messy, might pervert the course of an execution, and would make possible a whole new set of mistakes in addition to those possible in the original program and in the proof.

Apt's proof system allows references to shared auxiliary variables only in the global invariants, but it does require the use of auxiliary variables to represent process state completely. Using such a proof to direct tracing would be easier than using a Levin-Gries

$\{(\forall i: 1 \leq i \leq N: set_i = \{i\}) \wedge set_0 = \phi\}$
B::$\{M(0,0,set_0)\}$
  if $[]_{i=1:N}$ Least(i)?(m,set_0,set_i) $\rightarrow$ skip **fi**
  $\{M(m,N,set_0)\}$

$||$A:: $[||_{i=1:N}$ $\{set_i = \{i\}\}$ Least(i) $\{M(0,0,set_i)\}]$

$\{(\forall i: set_i = \{i\})\}$
Least(i)::
  $\{set_i = \{i\}\}$
  integer mymin, theirmin, mysize, theirsize;
  mymin, mysize := a_i, 1;
  $\{M(mymin,mysize,set_i)\}$

  do
  $[]_{j=1:N \wedge i \neq j}$ 0<mysize<N; Least(j)!(mymin,mysize,$set_i \cup set_j$, ϕ) $\rightarrow$ $\{M(mymin,0,set_i)\}$ mysize:=0

  $[]_{k=1:N \wedge i \neq k}$ 0<mysize<N; Least(k)?(theirmin,theirsize,set_i,set_j)
    $\rightarrow$ $\{M(min(mymin,theirmin),mysize+theirsize,set_i)\}$
      mymin,mysize:=min(mymin,theirmin),mysize + theirsize
  od;

  $\{M(mymin,mysize,set_i) \wedge (mysize=0 \vee mysize = N)\}$
  if mysize=0 $\rightarrow$ skip $\{M(mymin,0,set_i)\}$
  [] mysize=N $\rightarrow$ $\{M(mymin,N,set_i)\}$
    B!(mymin,set_i, ϕ)
    $\{M(mymin,0,set_i)\}$
  fi
  $\{M(mymin,0,set_i)\}$

$M(mn,size,S) \equiv (size = |S| \wedge (S=\phi \vee mn = \min_{j \in S}(a_j))$
UNION: $(\cup_{i=0:N} set_i) = 1:N \wedge (\forall i,j: 0 \leq i < j \leq N: set_i \cap set_j = \phi)$

Figure 3.2: Levin-Gries Annotation of Minset

proof, since all auxiliary variables could be maintained locally, but we would still have to modify the code of each process individually, leaving the door open for additional error. We also incur, as with any use of global state, the expense of checking all concurrent states to determine if the predicate is true at each.

3.5 A Traceable Proof System

What we would like is a proof system which, like Apt's, allows no nonlocal references in process annotations, but also which avoids global invariants and auxiliary variables, and highlights the role of causality. Just as we would like to trace what we assert, we would like to assert only what we can trace and analyze, and yet still be able to make assertions about all causal relationships.

A proof system like that of Misra and Chandy [MC81] or of Soundararajan [Sou84] might seem to satisfy our demands. They describe process executions by defining communication histories. The verifier annotates each process in isolation, without global references, and then describes inputs and outputs as assignments to communication history *traces*. Given these histories, the verifier can make assertions on them, rather than on program variables. The traces constitute, in effect, an invariant describing all process interaction. A parallel composition rule tells the verifier how to use the traces to draw nonlocal conclusions from process post–conditions.

Though these systems rely on reasoning about process state and interprocess communication rather than about global state, they are not satisfactory for our purposes because they stress the isolation of process proofs. We do not believe that it is useful to do process proofs in isolation. Rather, we believe that postconditions of blocking communication statements should be derived explicitly by considering the status of both the local process and the remote process involved in the communication. Proofs in isolation, like assertions about global state, ignore the crucial role of causality, and so do little to clarify our

reasoning or to make trace analysis more tractable.

If the proof systems of Apt and of Levin and Gries, with their use of global reasoning, lie at one end of the spectrum, these, with their extreme isolation of processes, lie at the other. We need to claim the middle ground where causal reasoning can play its proper part. What makes this difficult, if we want to make tracing easy, is that the auxiliary variables used in the proof systems discussed in the previous section do represent valuable aspects of process state.

Lamport showed in [Lam88b] that control predicates can replace auxiliary variables for the representation of process program counter values. Recall from page 25 that control predicates are expressions of the form $\mathrm{at}(\alpha)$, $\mathrm{in}(\alpha)$, and $\mathrm{after}(\alpha)$, where α is a program fragment. We say that $\mathrm{at}(\alpha)$ is true if control is at the entry point of α, $\mathrm{in}(\alpha)$ is true if control is at the entry point of or inside α, and $\mathrm{after}(\alpha)$ is true if control is at the exit point of α. If α is an atomic statement, $\mathrm{at}(\alpha) = \mathrm{in}(\alpha)$.

Unlike auxiliary variables, control predicates have a formal connection with program state. As the program counter of a process changes value, the value of the process's control predicates register its state, with no intervention by the verifier. Their use requires no redefinition of the semantics of CSP operations, since reference to them merely makes explicit what is already implicit in the state of the program.

Neither the Levin-Gries nor the Apt system allows assertions to contain control predicates. And, unfortunately, even if we allow their use, control predicates alone cannot represent the entire control state, as Apt has shown [Apt81]. Intuitively, we can see that the control state of a process involves more than the value of its program counter. The control state of a distributed process includes the position of the process's state in the program's causal order. It is not enough to say that control lies in a given program fragment. We must also be able to say which states have a causal relationship to the state in question. We must be able to place the process state in its historical context within the execution.

Auxiliary variables, as we can see in the proof systems we have considered, can record these causal relationships. An auxiliary variable can count how many messages have been sent from one process to another, or show, as in Minset, which processes have directly or indirectly communicated with a process, as a control predicate cannot.

There is, however, an implicit aspect of process state which, if used in conjunction with control predicates, does give us the power we need, and without the bad features of auxiliary variables: the process's vector clock value [Fid88, Mat89], which we defined on page 45. Vector time accurately records the causal order of process states in a distributed system. Vector clocks and control predicates used together completely define process control state.

Vector clocks, like control predicates and unlike auxiliary variables, have a formal connection with process state. They are updated by the progress of the process program counter, not by operations with which the verifier augments the code being annotated. They require no change to the semantics of CSP; when we use them, we only make explicit the causal relationships which interprocess communications, by their nature, impose.

When we verify, we want to show the necessary relationships between process states. Several researchers have pointed out the usefulness of vector clocks in debugging [Fid88, Che89, HMW89], but their ability to represent the causal order of program states also makes them valuable in a proof system for distributed systems. Because it represents the partial order, vector time must reflect all necessary causal relationships. Vector clocks can reflect the fact that, for instance, not only did T_1 and T_2 have value $<5,6,3>$ after a particular communication in a given run of some program $\mathcal{P}$, but, say, that in any run of $\mathcal{P}, T_{12} = T_{22} = T_{11} + 1$ after all executions of that send/receive pair, and further that in each such case $T_{11} = 2i + 1$, where i is a loop index in π_2.

For our purposes, though, it is necessary but not sufficient that vector clocks can be helpful in proofs of distributed programs. What especially interests us is that, in addition, vector clock values, unlike auxiliary variables, are readily traceable. Each process updates

its own vector clock, using only its local knowledge. No global knowledge is needed, yet the clocks faithfully record the partial order of process states. Since they are kept locally, we can log their value just as we log any other of the process's variables, without reference to the value of remote process state.

As a practical matter, we must implement sends and receives to update vector clocks correctly and efficiently, but since a synchronous communication requires at least an acknowledgment from the receiver, we can piggy-back the vectors onto the messages at little additional cost. Because all the required code can be inserted at compilation, the verifier does not have to change the program, so we avoid the problem of introducing errors during verification.

3.6 A Causal Proof System for CSP

In our proof system for CSP, the fragment of CSP we consider is essentially that discussed in Section 3.2. We do, however, make the additional assumption that a synchronous communication updates the vector clocks of the communicating processes appropriately.

We use no auxiliary variables in our proofs, only program variables, control predicates and vector clock values. Like Apt [AFDR80], we make only local references in our process annotations, and then use program invariants to relate process states. Unlike Apt's, our program invariants do not make assertions about global state. They use vector timestamps and control predicates to make pairwise comparisons of process states, just as we did when we discussed mutual exclusion in our example program. We call our program invariants *glue* predicates, because we use them to tie together the local states of two processes.

We use glue predicates only to associate the preconditions and postconditions of a pair of input and output statements. Suppose S is a parallel program $\pi_i :: S_1 \parallel \ldots \parallel \pi_n :: S_n$. For each pair of potentially communicating input and output statements S_i' and S_j', where S_i' is a component of S_i and S_j' is a component of S_j, we must assert $glue(S_i', S_j')$. Only

the variables and vector clocks defined in π_i and π_j and the statement labels of S_i', S_j', π_i and π_j may be free in $glue(S_i', S_j')$.

We typically use glue predicates of the form

$$glue(S_i', S_j') = (state_i = \sigma_i) \supset (control\_state_j = \tau_j' \supset state_j = \tau_j),$$

where σ_i and τ_j are process states, $state_i$ is a function of process π_i's state, $control\_state_j$ is a function of π_j's control predicates and vector clock, and $state_j$ is a function of π_j's state. Such a glue predicate tells us that, if process π_i is in some given state σ_i, then if π_j is in the proper control state τ_i', it must be in state τ_i.

When we write a proof of a distributed program, glue predicates let us assert necessary relationships between process states so that we can show that the causal relationships defined in the specification hold. Then, as we test, we can first record local traces by logging the variables referenced in process annotations, and then use the glue predicates to show us which trace records we should compare to test whether the expected causal relationships did in fact occur. In the proof, the glue predicates help us derive the postconditions of communication commands. In the trace, they tell us how to relate process traces so that we can detect the properties in which we are interested without having to make unnecessary state comparisons.

A glue predicate is universally true, but its truth is vacuous except at those points at which the process states mentioned in its antecedent clauses hold. Since the local states of two processes can be causally related only through communication, there is no need to link the pre- and post-conditions of local statements in different processes. When an assignment causes a state transition, the transition is purely local; we do not need to refer to the state of any other process to understand what happened. When processes communicate, however, causality flows between the processes and we need to look at both to understand their transitions.

The causal relationship, of course, may be indirect. Two processes may communicate

by way of one or more intermediaries, and we may want to assert something about the states of the processes at the ends of the chain of communications. We can derive a glue predicate for the indirectly communicating pair of communication statements from the glue predicates of the directly communicating statements which link them. Recall that the j^{th} element in process π_i's vector clock T_i identifies the latest significant state transition by process π_j which can influence π_i. We can use this fact to assert in a glue predicate that some relationship exists between the states of π_i and π_j if the j^{th} elements in the vector clocks of the two processes are equal. Suppose that $T_{ij} = 3$ after π_i receives from π_k. Then π_i's state is influenced by the state of π_j at the point at which π_j sent the message which incremented T_{jj} to 3. We could relate these two states by asserting in a glue predicate $glue(s_j, r_i)$ that both T_{ij} and T_{jj} have value 3.

With this background, we can define our causal proof system for CSP. We assume that we have a valid deductive system for the data types and operations of CSP, as we explain more fully in the following chapter. If we can derive q from p in this system, we write $p \vdash q$. A proof rule is defined by placing the premises over a line and the conclusions beneath the line. Thus

$$\frac{p, q}{r}$$

means that if we can conclude p and q using the axioms and rules of our system, then we can also conclude r. In each rule or axiom, if p is a pre- or post-condition of S_i, where S_i is a subprogram of process π_i, then only the vector clock and variables of π_i may appear free in p. An assertion p may also reference variables existentially quantified over the program annotation. These serve as placeholders for constant, but unknown, values, such as the value of a vector timestamp.

Our proof system consists of the following axioms and proof rules.

- Axioms

  1. Preservation: $\{p\}S_i\{p\}$ where no free variable of p is subject to change in S

2. Assignment: $\{p_e^x\}x := e\{p\}$

3. Input: $\{p\}\pi_j?x\{true\}$

4. Output: $\{p\}\pi_j!e\{true\}$

- Rules

1. Sequence: $\dfrac{\{p\}S_i\{q\}, \{q\}S_i'\{r\}}{\{p\}S_i; S_i'\{r\}}$

2. Alternation: $\dfrac{(\forall j : \{p \wedge b_j\}S_i^j\{q\})}{\{p\}\textbf{if } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ fi}\{q\}}$

3. Repetition: $\dfrac{(\forall j : \{p \wedge b_j\}S_i^j\{p\})}{\{p\}\textbf{do } []_{j=1:n} b_j \rightarrow S_i^j \textbf{ od}\{p \wedge (\forall j : \neg b_j)\}}$

4. Consequence: $\dfrac{p \vdash p_1, \{p_1\}S_i\{q_1\}, q_1 \vdash q}{\{p\}S_i\{q\}}$

5. Satisfaction:

$$\frac{\begin{array}{l}\{p\}c_i\{true\},\ p \vdash T_i = \vec{X},\\ \forall c_j : c_i \text{ matches } c_j \wedge \{q\}c_j\{r\} \wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i)) :\\ \qquad (p \wedge q \wedge glue(c_i, c_j)) \vdash (s \wedge glue(c_i, c_j))_{e, T_i', T_j'}^{x, T_i, T_j}\end{array}}{\{p\}c_i\{s\}}$$

where c_i in π_i and c_j in π_j are communication commands, c_i matches c_j if and only if c_i and c_j are syntactically matching send and receive commands, only the variables and vector clocks defined in π_i and π_j and the statement labels of c_i, c_j, π_i and π_j are free in $glue(c_i, c_j)$, and $T_i' = T_j' = synch(i, j, T_i, T_j)$.

Most of these axioms and rules are similar to those used in the other proof systems for CSP, but the communication axioms and satisfaction rule deserve discussion. In the other proof systems, a combining meta-rule ensures that the verifier return to the process annotations to write a satisfaction or cooperation proof that justifies any arbitrary postassertions derived in isolation using input and output axioms. Our procedure is different. Our satisfaction rule is not a meta-rule to be applied to completed process annotations. It is

a regular proof rule used to derive those annotations in the first "pass" that the verifier makes through the code.

Our input and output axioms do not let us make arbitrary "miraculous" postassertions. For partial correctness, if S is a nonterminating statement, then $\{true\}\ S\ \{true\}$ is a tautology [Gri81]. To assert "true" as the postassertion of a send or receive is but to put a stop sign in the annotation. Any preassertion of the command justifies "true" as its postassertion, yet "true" keeps us from deriving any unwarranted postassertions of the statement which follows. To say anything useful about process state when a communication command terminates, the verifier must use the satisfaction rule to derive the appropriate postassertion.

We force the verifier to use the satisfaction rule during the annotation of a process, rather than after she has finished her process annotations, to focus her attention on the causal relationships between process states. Using the rule forces her to discover and use the pairwise causal relationships between communicating processes to derive her assertions.

The satisfaction rule is complicated looking, but its meaning is not hard to understand. It says that what we can assert as the postcondition of a synchronous communication command depends on the preassertions of any commands with which it could communicate, the glue invariants which link the potentially communicating pairs, and the command's own preassertion. The conjunct

$$\neg(\vdash (\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i))$$

means "it cannot be shown that $(\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i)$." A send and receive can communicate only if the process states of the sender and receiver at the entry points to the communication commands are concurrent. If we cannot show from our annotation that one of a pair of syntactically matching communication commands necessarily precedes the other, then we must consider the effects of their communication when we derive their postassertions. If a send or receive cannot communicate with any other, the satisfaction

rule will not apply. If there is no remote communicating command, then we can derive no meaningful postassertion, which is proper since the command in isolation will not terminate.

Our axioms and proof rules use no global reasoning. Only in the satisfaction rule are two processes implicated, and there the association is between just the two processes, at points of communication. Since no process annotation references nonlocal variables, we do not need to carry out a noninterference proof. This makes proving programs correct with our system less complicated than doing so with the systems of Levin and Gries or Schlichting and Schneider.

3.7 Two Examples: MUTEX and Minset

We will illustrate our proof system with two examples. First, we give an annotation of the simple program for token-based mutual exclusion used as an example in Section 2.2. We also show that this program is a correct implementation of the specification for a grant/release program given in Chapter 2. Then, we give an annotation of the program Minset introduced earlier in this chapter to show that even so complicated a program is amenable to causal reasoning.

3.7.1 Annotation of MUTEX

Using our proof system, we could give an annotation for the toy program MUTEX like that in Figure 3.3. The timestamp values $\vec{W}$, $\vec{X}$, $\vec{Y}$, and $\vec{Z}$ are existentially quantified over the annotation of MUTEX. The postassertions of the send and receive statements

π_i:: do [] $\{T_i = \vec{W}\}$ α_i : $\pi_{(i+N-1) \bmod N}$?token → $\{T_i = \vec{X} > \vec{W}\}$
β_i : if want_cs$_i$ → critsec$_i$; want_cs$_i$:= false
[] not(want_cs$_i$) → skip
fi
$\{T_i = \vec{Y} \geq \vec{X}\}$
γ_i : $\pi_{(i+1) \bmod N}$!token
$\{T_i = \vec{Z} > \vec{Y}\}$
[] true → do_other$_i$
od;

Glue: $(after(\alpha_i) \wedge T_i = \vec{X}) \supset (after(\gamma_{(i+N-1) \bmod N}) \supset T_{(i+N-1) \bmod N} = \vec{X})$

Figure 3.3: An annotation of the toy program MUTEX.

are justified by the satisfaction rule. For the postcondition of the receive, we have

$$
\begin{aligned}
[T_i = \ & \vec{W} \wedge T_{(i+N-1) \bmod N} = \vec{Y} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))] \\
\vdash [T_i \ & = \vec{X} > \vec{W} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \\
& \quad \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))]^{\text{token},T_i,T_{(i+N-1) \bmod N}}_{\text{token},T'_i,T'_{(i+N-1) \bmod N}}
\end{aligned}
$$

where T'_i and $T'_{(i+N-1) \bmod N}$ are the updated vector clocks, which is equivalent to

$$
\begin{aligned}
[T_i = \ & \vec{W} \wedge T_{(i+N-1) \bmod N} = \vec{Y} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))] \\
\vdash [T'_i \ & = \vec{X} > \vec{W} \wedge \\
& ((after(\alpha_i) \wedge T'_i = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T'_{(i+N-1) \bmod N} = \vec{X}))]
\end{aligned}
$$

which is true. Similarly, for the postcondition of the send, we have

$$
\begin{aligned}
[T_i = \ & \vec{Y} \wedge T_{(i+1) \bmod N} = \vec{W} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))] \\
\vdash [T_i \ & = \vec{Z} > \vec{Y} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \\
& \quad \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))]^{\text{token},T_i,T_{(i+1) \bmod N}}_{\text{token},T'_i,T'_{(i+1) \bmod N}}
\end{aligned}
$$

which is equivalent to

$$
\begin{aligned}
[T_i = \ & \vec{Y} \wedge T_{(i+1) \bmod N} = \vec{W} \wedge \\
& ((after(\alpha_i) \wedge T_i = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T_{(i+N-1) \bmod N} = \vec{X}))] \\
\vdash [T_i' \ & = \vec{Z} > \vec{Y} \wedge \\
& ((after(\alpha_i) \wedge T_i' = \vec{X}) \vdash (after(\gamma_{(i+N-1) \bmod N}) \vdash T'_{(i+N-1) \bmod N} = \vec{X}))]
\end{aligned}
$$

which is true. The timestamp value in the precondition of the send is greater than or equal to $\vec{X}$ because π_i may have executed sends or receives in its critical section.

The annotation and glue predicate taken together imply that for any two critical section executions *critsec* and *critsec'*, either *in(critsec)*→*in(critsec')* or *in(critsec')*→*in(critsec)*, since $T < T' \Leftrightarrow e \rightarrow e'$, where T is the timestamp of e and T' is the timestamp of e'. This guarantees mutual exclusion.

As we noted in Section 2.2.3, we can also see MUTEX as an implementation of the specification given in Figure 2.2. If we assume that some process starts MUTEX running with a parallel command which represent the specification's program counter *INIT*, then we can show that MUTEX correctly implements the specification. State function *request* is set in do_other_i in the assignment of value true to $want\_cs_i$. Its value is the number i of the process π_i which sets the boolean.

State function *REQ* is implemented in MUTEX in the selection of the guard $want\_cs_i$ in statement β_i. The state function q is distributed throughout the processes. Process π_i "joins the queue" when it selects $want\_cs_i$. This also nulls *request*.

MUTEX implements *GRANT* as the statement sequence $critsec_i$. The process number of the process in its critical section implements state functions *granted* and *grantarg*. Entering the critical section removes a process from the queue of those requesting access and sets *granted* and *grantarg*. The statement which assigns false to $want\_cs_i$ represents *REL*. This statement nulls *granted* and *grantarg*.

We note in passing that the flow of control in MUTEX also guarantees that the liveness properties of the specification are satisfied. Arbitrarily long delays are possible while

neighboring processes approach synchronization, but deadlock is not. A more thorough discussion of liveness is beyond the scope of this thesis.

Each assertion in the annotation refers only to a process's locally maintained vector timestamp value, so we could efficiently log individual process traces based on this annotation, and then analyze them to see if the state functions assume the values that we expect. In other, more realistic, programs, we would make more assertions about process data values and perhaps have syntactically matching input and output statements which do not communicate. Our glue predicates would have to relate the data values in the pre- and post-assertions of the communicating commands, and the application of the satisfaction rule would be less trivial. In principle, though, our procedure would be the same. We would annotate the code, record in our traces the variables referenced in our assertions, and analyze the traces to see if the desired causal relationships held.

3.7.2 Annotation of Minset

We now will briefly sketch a proof of the partial correctness of program Minset. We want to show that if B terminates, it will have the minimum value of A stored in its variable B.

In Figure 3.4 we show an annotation of Minset which includes local and glue predicates. The sequential reasoning in the process proofs of Least(i) and of B is straightforward. The postconditions of the sends and receives must be derived using the satisfaction rule. W and $\vec{X}$ are existentially quantified over the program annotation. We use subscripts on statement labels and variables when necessary to indicate to which process they belong. We define (Nm:P(m)) to equal the number of elements m in a clock vector for which P(m) is true; note that if T_{ik} is nonzero, then π_i and π_k have communicated, and that variable mysize keeps track of how many members of A's set Least(i) is responsible for.

The first glue predicate is $glue(\alpha_i, \beta_j)$. It asserts that until Least(i) sends Least(j) its minimum, no process has communicated with both Least(i) and Least(j). If this is so, the

Least(i)::
integer mymin, theirmin, mysize, theirsize;
mymin, mysize := a_i, 1;
$\{a_i = \text{mymin} \wedge \text{mysize} = 1\}$
do
$[]_{j=1:N \wedge i \neq j}$ 0<mysize<N;
$\{1 \leq j \leq N \wedge a_i \geq \text{mymin} \wedge [\ (\text{mysize}=1 \wedge T_i = \vec{0}) \vee ((\text{Nm}:T_{im} \neq 0) = \text{mysize})\ ]\ \}$
α: Least(j)!(mymin,mysize) $\rightarrow$
$\{a_i \geq \text{mymin}\}$
mysize := 0
{mysize=0}

$[]_{k=1:N \wedge i \neq k}$ 0<mysize<N;
$\{a_i \geq \text{mymin} \wedge [\ (\text{mysize}=1 \wedge T_i = \vec{0}) \vee ((\text{Nm}:T_{im} \neq 0) = \text{mysize})]\}$
β: Least(k)?(theirmin,theirsize) $\rightarrow$
$\{a_i \geq \text{min}(\text{mymin},\text{theirmin}) \wedge (\text{Nm}: T_{im} \neq 0) = \text{mysize}+\text{theirsize}\ \}$
mymin, mysize := min(mymin,theirmin), mysize + theirsize
$\{a_i \geq \text{mymin} \wedge (\text{Nm}: T_{im} \neq 0) = \text{mysize}\ \}$
od;
$\{(\text{mysize}=0) \vee ((\text{Nm}: T_{im} \neq 0)=\text{mysize}=N))\}$
if mysize=0 $\rightarrow$ skip {mysize=0}
[] mysize=N $\rightarrow$
$\{(\text{Nm}: T_{im} \neq 0)=\text{mysize}=N \wedge a_i \geq \text{mymin}\}$
γ: B!mymin
$\{(\text{Nm}: T_{im} \neq 0)=N+1 \wedge a_i \geq \text{mymin}\}$
fi

B:: **if** $[]_{i=1:N}$ δ: Least(i)?m $\rightarrow$ skip **fi**
$\{1 \leq i \leq N \wedge m=W \wedge ((\text{Nm}: T_{(N+1)m} \neq 0)=N+1)\}$

Glue predicates:
1) $glue(\alpha_i, \beta_j) \equiv [\text{at}(\beta_j) \wedge T_j = \vec{X} \wedge T_{ji} = 0] \vdash [\text{at}(\alpha_i) \vdash (T_i \cdot \vec{X} = 0)]$
2) $glue(\alpha_l, \beta_j) \equiv [\text{after}(\beta_j) \wedge \text{min}(\text{mymin}_j,\text{theirmin}_j) = Y \wedge T_j = \vec{Z}]$
$\vdash [\forall l\text{: } (\text{after}(\alpha_l) \wedge T_{jl} = \vec{Z}_l \neq 0) \vdash \text{mymin}_l \geq Y)]$
3) $glue(\gamma_i, \delta) \equiv [\text{after}(\delta) \wedge m=W] \vdash [\text{after}(\gamma_i) \supset \text{mymin}_i = W)]$

Figure 3.4: A Causal Annotation of Minset

dot product of the clock vectors of the sender and receiver will be 0, since where one has a nonzero element, the other will have a zero element. We assert that $T_{ji} = 0$ to ensure that π_j has not yet communicated with π_i.

The second glue predicate, $glue(\alpha_l, \beta_j)$, relates Least(j)'s state at the exit point of its receive to the states of those Least processes which have communicated with it directly or indirectly, at the exit points of their sends. We identify the processes whose values Least(j) knows by seeing which elements in Least(j)'s vector clock are nonzero. We assert that at the exit point of the receive, the receiver's new minimum will be less than or equal to the minimum sent by every such process. We know that this is true even when communication was indirect because each time a Least process receives a new value, it sets its minimum to the lesser of the incoming value and its current minimum before it communicates with anyone else.

In the third glue predicate, $glue(\gamma_i, \delta)$, we assert that after the communication between a Least(i) and B, B's variable m and the Least(i)'s minimum have the same value. Since only one Least process communicates with B, we don't need to use vector time here. Control variables adequately define control state in this case.

Using these glue predicates, we must demonstrate satisfaction to remove the "true" postconditions in the process proofs. It must be the case that

$$(pre(\alpha) \wedge pre(\beta) \wedge glue(\alpha_i, \beta_j)) \vdash$$
$$(post(\alpha) \wedge post(\beta) \wedge glue(\alpha_i, \beta_j))^{\mathrm{theirmin}_j, \mathrm{theirsize}_j, T_i, T_j}_{\mathrm{mymin}_i, \mathrm{mysize}_i, T_i', T_j'}$$

where i is the process index of the sender, j that of the receiver, and $T_i' = T_j' = synch(i, j, T_i, T_j)$. We must also establish

$$(pre(\gamma) \wedge pre(\delta) \wedge glue(\gamma_i, \delta)) \vdash$$
$$(post(\gamma) \wedge post(\delta) \wedge glue(\gamma_i, \delta))^{\mathrm{m}, T_i, T_j}_{\mathrm{mymin}, T_i', T_j'}.$$

T_i' and T_j' are as we defined them above. We note that, taken together, the assertion in the first glue predicate that no process has talked to both sender and receiver, and the

way in which vector clocks are synchronized, let us state in $post(\beta)$ that the number of nonzero elements in the receiver's vector clock is equal to the sum of mysize and theirsize. The rest of the reasoning used in showing satisfaction is straightforward, albeit tedious, and we leave it to the reader.

Given the process and satisfaction proofs, we can see that B will have the minimum value if it terminates. When a Least(i) sends B its value, its vector clock will have N nonzero elements. From the second glue predicate, we know that this means that the value it sends is the minimum of the N values.

3.8 The Value of the Proof System

Though our proof rules and axioms are somewhat similar to those previously proposed for CSP, our proof system is unique in its emphasis on the role of causality and in that it permits effective execution tracing of CSP programs. Other proof systems use constructs which are not readily traceable or not amenable to effective postmortem analysis, or which ignore the importance of causality in distributed processing.

If we attempt to use the systems which use auxiliary variables and non-local reasoning to tell us what to trace, we must reshape our programs into a form which allows tracing. This will probably mean that we must create and maintain new variables which stand for the proof's auxiliary variables. Even worse, we may have to add message passing and synchronization to validate the proof's non-local assertions. The alternative would be to forget the proof and use some testing strategy which makes little use of the proof assertions. Neither choice is desirable. The first leads us to abandon the program; the second, to abandon the work we did writing the proof. Our proof system lets us benefit from the proof when we test, and learn from testing as we devise a valid proof.

We also believe that since thinking about causation is the key to understanding distributed programs, it is better to write causal proofs than ones which rely on global

assertions or which treat processes in isolation. It is easier to evaluate the effect of one statement or a pair of statements than to consider some overall state of the system. Apt *et al.* argue that their cooperation proofs reduce the annotation of a CSP process to local reasoning, but this ignores the fact that their invariants are global and unrestricted in content. Soundararajan, on the other hand, focuses so strictly on local state in his process annotations that he cannot consider the state of a remote communicating process. He would have us postpone consideration of the effects of communication until every process has been annotated and a communication invariant constructed, rather than consider communicating sends and receives pairwise, as we come to them in the course of annotation.

It is probably the case that no proof system can make it easy to come up with appropriate assertions for a complex program like Minset, but the restrictions we place on our annotations and the way we exploit causality do seem to make the job more tractable.

Chapter 4

Soundness and Completeness of the CSP System

In this chapter we show that our proof system for CSP is sound (that what we can prove in it is true), and that, relative to some complete deductive system for the data types and operations of CSP, the system is complete (that we can prove anything that is true.) To show the soundness and completeness of the system, we mostly follow the style of reasoning used by Owicki in her proof of the soundness and completeness of her parallel programming language [Owi75, Owi76]. Our reasoning is also influenced by the proofs offered by Apt in [ABM79], [Apt81] and especially [Apt83]. The proofs given in this chapter treat partial correctness only.

First, in Section 4.1, we give a proof of a simple sequential program to illustrate just what we mean when we talk about a formal program proof. Then, in Section 4.2, we take an operational approach to describe the semantics of CSP more formally than we did in Section 3.2. In Section 4.3 we approach the soundness of our axiomatic description of CSP from the last chapter from two directions. Finally, in Section 4.4, we prove that the axioms and proof rules are relatively complete.

4.1 An Example of a Formal Program Proof

We discussed in Chapter 3 how we use proof outlines to make showing correctness easier, but to understand some of the proofs of theorems that we present in Sections 4.3 and 4.4, we need to be familiar with the structure of a formal program proof of partial correctness. We show in Figure 4.1 a proof adapted from Owicki [Owi75]. The sequential program POWER computes the function $z = x^y$ for integers x and y, if $y \geq 0$. We want to prove

$$\{y \geq 0\} POWER \{z = x^y\}.$$

We give first a proof outline for POWER, and then a formal proof. In the latter, lines 1 through 4 describe the initialization of variables "z" and "temp." Lines 6 through 10 describe the **while** loop, while line 11 describes the effect of the program's statements taken together.

We see that, compared to the proof outline, such a formal proof is very tedious to write, even for such a simple program as POWER. That is why we normally use the more informal annotations. Nonetheless, it is this formality that undergirds our reasoning when we derive our proof outlines, and that lets us show in this chapter that our proof system is sound and relatively complete.

4.2 An Operational Semantics for CSP

Now, following Apt [Apt83], we define an operational semantics for our CSP subset. Our fragment of CSP is one whose expressions are made from nonlogical symbols of an arbitrary first-order language L with equality. The definition of its semantics is that of Apt, except that

- we define how vector timestamps are updated on a synchronous communication;

POWER:: $\{y \geq 0\}$
begin z := 1;
temp := y;
$\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$
loop:: **while** temp > 0 **do**
$\{ \text{temp} > 0 \wedge z = x^{y-\text{temp}} \}$
begin z := x * z;
temp := temp −1
$\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$
end
$\{ \text{temp} = 0 \wedge z = x^{y-\text{temp}} \}$
end
$\{ z = x^y \}$

1. $\{y \geq 0 \wedge 1 = 1\}$ z := 1 $\{y \geq 0 \wedge z = 1\}$ by the assignment axiom;
2. $\{y \geq 0\}$ z := 1 $\{y \geq 0 \wedge z = 1\}$ by step 1 and the consequence rule, using $y \geq 0 \vdash (y \geq 0 \wedge 1 = 1)$;
3. $\{y \geq 0 \wedge z = 1 \wedge y = y\}$ temp := y $\{y \geq 0 \wedge z = 1 \wedge \text{temp} = y\}$ by the assignment axiom;
4. $\{y \geq 0 \wedge z = 1\}$ temp := y $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$ by step 3 and the consequence rule;
5. $\{ \text{temp} - 1 \geq 0 \wedge x * z = x^{y-(\text{temp}-1)} \}$ z := x * z $\{ \text{temp} - 1 \geq 0 \wedge z = x^{y-(\text{temp}-1)} \}$ by the assignment axiom;
6. $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \wedge \text{temp} \geq 0 \}$ z := x * z $\{ \text{temp} - 1 \geq 0 \wedge z = x^{y-(\text{temp}-1)} \}$ by step 5 and the consequence rule;
7. $\{ \text{temp} - 1 \geq 0 \wedge z = x^{y-(\text{temp}-1)} \}$ temp := temp −1; $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$ by the assignment axiom;
8. $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \wedge \text{temp} > 0 \}$
begin $\bar{z}$:= x * z; temp := temp −1 **end**
$\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$
by steps 6 and 7, and the sequence rule;
9. $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$ *loop* $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \wedge \neg(\text{temp} > 0) \}$ by step 8 and the repetition rule;
10. $\{ \text{temp} \geq 0 \wedge z = x^{y-\text{temp}} \}$ *loop* $\{ z = x^y \}$ by step 9 and the consequence rule;
11. $\{ y \geq 0 \}$ POWER $\{ z = x^y \}$ by steps 2, 4 and 10, and the sequence rule.

Figure 4.1: A formal proof of partial correctness for POWER.

- we define a global state σ to be equivalent to $\sigma_1 \cup \sigma_2 \cup \ldots \cup \sigma_n$, for disjoint process states $\sigma_1 || \sigma_2 || \cdots || \sigma_n$, where $\sigma_i || \sigma_j$ if and only if $(\sigma_i(T_{i,i}) \not< \sigma_j(T_{j,i})) \wedge (\sigma_j(T_{j,j}) \not< \sigma_i(T_{i,j}))$

where $\sigma_i(x)$ is the value of x in state σ_i. Our goal is to tailor Apt's semantics to fit our understanding of the nature of distributed processing, but to leave it essentially the same so that our results and his are comparable. Apt does not consider vector time, and assumes that the global program state σ, rather than the process state σ_i, is fundamental. With our orientation towards processes and causation, it makes more sense to say that any "global" state is but the union of the individual, and strictly disjoint, process states. In either case, a program state σ is a function that assigns values to all variables of L from the domain of its interpretation, J. More intuitively, for us, as for Apt, σ is merely a global snapshot after an atomic operation.

In what follows we assume some arbitrary interpretation J which assigns to L's non-logical symbols appropriate relations or functions over its domain, so we write $\models p(\sigma_i)$ rather than $\models_J p(\sigma_i)$ to indicate that formula p is true in state σ under J. We write $\models p$ if p is true (under the arbitrary interpretation) for all σ.

As in Chapter 3, the notation p_e^x indicates the substitution of e for each free occurrence of x in p. We write σ_e^x to indicate the state which is equivalent to σ except for the value of x, which has value e in σ_e^x. As Gries shows in [Gri81, Lemma 4.6.2], $p_e^x(\sigma) = p(\sigma_e^x)$.

Our semantics uses the $\rightarrow$ and $\rightarrow_k^h$ relations between statement/state pairs defined by Apt. (Do not confuse these relations with Lamport's "happened before" relation, also designated by $\rightarrow$.) Intuitively, if we have

$$<S_i, \sigma> \rightarrow <S_i', \tau>,$$

we mean that if statement S_i in process π_i executes one step beginning in global state σ, then its execution can lead nondeterministically to state τ, with S_i' the remaining part of S_i to be executed by π_i. We say "nondeterministically" because in a distributed program

it may be the case that some statement S_j in π_j may be able to execute in σ also. Whether S_i or S_j is next in some global ordering is arbitrary.

We also say

$$<S_1||\cdots||S_N,\sigma> \rightarrow_k^h <S_1'||\cdots||S_N',\tau>$$

where h is a *history* and k a natural number. A history is a sequence of *records of communication*, that is, triples (a,i,j), where a is the value sent from π_i to π_j in a synchronous communication. By this notation we mean that beginning in state σ, we can execute parallel programs $S_1||\cdots||S_N$ and reach state τ in k steps with all interprocess communications recorded in h, with $S_1'||\cdots||S_N'$ left to execute.

To define the meaning of a statement, we use Apt's relation $\mathcal{M}(S)(\sigma)$, which maps a statement S executed in a state σ to a set of states. State σ maps each program variable to a value. Executing S in σ effects a transition from σ to some state τ, which also maps variables to their values. τ must be an element of the set of states defined by $\mathcal{M}$. Thus, we define the meaning of an assignment statement "$x := e$" executed in state σ, $\mathcal{M}(x := e)(\sigma)$, to be the set of states which results from determining the reference represented by variable x and the value of expression e in σ, and storing the value in the reference. This set, clearly, contains just one element, σ_e^x.

We define the empty statement E such that $E;S_i \equiv S_i;E \equiv S_i$. If S_i terminates in τ, $S_i' \equiv E$. Recall that in an alternation or a repetition statement in process π_i, b_j represents the statement's jth boolean guard and S_i^j the statement guarded by that boolean.

We formally define $\rightarrow$ and $\rightarrow_k^h$ as follows:

1. $<S_i,\sigma> \rightarrow <S_i',\tau>$, for statements S_i and S_i' of process π_i, and global program states σ and τ, is:

   (a) $<\textbf{skip},\sigma> \rightarrow <E,\sigma>$.

   (b) $<x := e,\sigma> \rightarrow <E,\mathcal{M}(x := e)(\sigma)>$.

   (c) $<\textbf{if } []_{j=1;m} b_j \rightarrow S_i^j \textbf{ fi},\sigma> \rightarrow <S_i^k,\sigma>$ if $\models b_k(\sigma)$ $(1 \leq k \leq m)$.

(d) $<\mathbf{do}\ []_{j=1:m} b_j \rightarrow S_i^j\ \ \mathbf{od}, \sigma> \rightarrow <S_i^k; \mathbf{do}\ []_{j=1:m} b_j \rightarrow S_i^j\ \mathbf{od}, \sigma>$ if $\models b_k(\sigma)$ $(1 \leq k \leq m)$.

(e) $<\mathbf{do}\ []_{j=1:m} b_j \rightarrow S_i^j\ \mathbf{od}, \sigma> \rightarrow <E, \sigma>$ if $\models \neg b_j(\sigma)$ for $j = 1, \ldots, m$.

2. $<S_1 || \cdots || S_N, \sigma> \rightarrow_k^h <S_1' || \cdots || S_N', \tau>$ for parallel programs $S :: S_1 || \cdots || S_N$ and $S' :: S_1' || \cdots || S_N'$, is:

(a) $<S_1 || \cdots || S_N, \sigma> \rightarrow_0^\epsilon <S_1 || \cdots || S_N, \sigma>$.

(b) If $<S_i, \sigma> \rightarrow <S_i', \tau>$, $(1 \leq i \leq N)$, then

$$<S_1 || \cdots || S_{i-i} || S_i || S_{i+1} || \cdots || S_N, \sigma> \rightarrow_1^\epsilon <S_1 || \cdots || S_{i-1} || S_i' || S_{i+1} || \cdots || S_N, \tau>.$$

(In this case, S_i alone executed one step locally.)

(c) If $S_i \equiv \pi_j ! e$ and $S_j \equiv \pi_i ? x$, then

$$<S_1 || \cdots || S_N, \sigma> \rightarrow_1^{(e,i,j)} <S_1' || \cdots || S_N', \mathcal{M}(x, T_i, T_j := e, T_i', T_j')(\sigma)>,$$

where $S_i' \equiv S_j' \equiv E$ and $S_k' \equiv S_k, k \neq i, j$, and T_i' and T_j' are both equal to

$$\begin{aligned} \text{synch}(i, j, \sigma(T_i), \sigma(T_j)) = \ & (\sup([\sigma(T_{i,1}), \ldots, \sigma(T_{i,i}) + d, \ldots, \sigma(T_{i,N})], \\ & [\sigma(T_{j,1}), \ldots, \sigma(T_{j,j}) + d, \ldots, \sigma(T_{j,N})]) \end{aligned}$$

where $d > 0$. (In this case π_i and π_j executed a synchronous communication, which acts as a distributed assignment statement and updates the vector timestamps T_i and T_j.)

(d) If $<S_1 || \cdots || S_N, \sigma> \rightarrow_k^h <S_1' || \cdots || S_N', \tau>$, then for any S_i'',

$$<S_1 || \cdots || S_i; S_i'' || \cdots || S_N, \sigma> \rightarrow_k^h <S_1' || \cdots || S_i'; S_i'' || \cdots || S_N', \tau>.$$

(This handles program composition.)

(e) If $<S, \sigma> \rightarrow_{k_1}^{h_1} <S', \sigma_0>$, and $<S', \sigma_0> \rightarrow_{k_2}^{h_2} <S'', \tau>$, then

$$<S, \sigma> \rightarrow_{k_1+k_2}^{h_1 \circ h_2} <S'', \tau>,$$

where $\circ$ denotes concatenation.

We can define the meaning of a CSP program $S :: S_1||\cdots||S_N$ by defining $\mathcal{M}(S)(\sigma)$ to be equivalent to

$$\{\tau : \exists k \geq 0, h \text{ such that } <S_1||\cdots||S_n, \sigma> \rightarrow_k^h <E||\cdots||E, \tau>\}.$$

We, like Apt [AFDR80], consider only finite processes. A nonterminating loop constitutes a semantic error. We could, however, make our proof system relatively complete for arbitrary processes by having vector time record changes in local state caused by local operations as well as those state transitions caused by interprocess communication.

As Apt observes, we need no special case in this semantics for either the send or the receive statement in isolation. Using the definition of the meaning of a CSP program, we see that for a send or receive statement α in isolation, we have

$$\mathcal{M}(\alpha)(\sigma) = \emptyset,$$

which fits with our intuitive understanding that a synchronous communication statement executed in isolation cannot terminate. Similarly, we do not need a special case for a repetition statement with all guards false, since according to the definition of CSP, such a statement fails.

4.3 Proof of Soundness

We can define the interpretation of the deductive primitive for our proof system in terms of the operational semantics. For statement S_i of process π_i in parallel program S, $\{p\}S_i\{q\}$ is equivalent to

$$\forall\, \sigma, \tau, k \geq 0, h : [(p(\sigma) \wedge \quad <S_1 \parallel \ldots \parallel S_i \parallel \ldots S_N, \sigma> \rightarrow_k^h$$
$$<S_1' \parallel \ldots \parallel \mathit{after}(S_i, \pi_i) \parallel \ldots \parallel S_N', \tau>) \vdash q(\tau)].$$

We define the *after* relation in a way similar to Apt.[1] By $\mathit{after}(S_i, \pi_i)$ we mean the remaining part of π_i to be executed after the execution of statement S_i. The second argument

[1] Unfortunately, here, as with the $\rightarrow_k^h$ relation, there is a conflict between Apt's notation and Lamport's. We must be careful to distinguish Apt's $\mathit{after}(S_i, \pi_i)$ relation from Lamport's $\mathit{after}(\alpha)$ control predicate.

may name either a process π_i or a statement S_i^l in π_i. We also define $before(S_i, \pi_i) \equiv S_i; after(S_i, \pi_i)$. Formally, we define $after(S_i, \pi_i)$ to be:

1. if $S_i \equiv \pi_i$, then $after(S_i, \pi_i) \equiv E$;

2. if π_i is **if** $[]_{j=1:m} b_j \rightarrow S_i^j$ **fi**, and S_i is a statement of S_i^j, $1 \leq j \leq m$, then

$$after(S_i, \pi_i) \equiv after(S_i, S_i^j);$$

3. if π_i is **do** $[]_{j=1:m} b_j \rightarrow S_i^j$ **od**, and S_i is a statement in S_i^j, $1 \leq j \leq m$, then

$$after(S_i, \pi_i) \equiv after(S_i, S_i^j); \pi_i;$$

4. if π_i is $S_i^1; \ldots; S_i^m$, then if S_i is a statement in S_i^k, $1 \leq k < m$, then

$$after(S_i, \pi_i) \equiv after(S_i, S_i^k); S_i^{k+1}; \ldots; S_i^m$$

else S_i is a statement in S_i^m and

$$after(S_i, \pi_i) \equiv after(S_i, S_i^m).$$

We observe that the first item in the definition handles simple cases like assignment, skip, input, output and terminating repetition statements. The other three items handle the more complicated cases in which S_i is a statement within a guarded command in an alternation or repetition statement, or within a statement which is a constituent of a sequence of statements.

As we noted in the previous chapter in our informal description of CSP, our semantics excludes the input–output guarded selection and repetition commands, since we are considering only partial correctness, and the CSP loop exit convention, in favor of explicit loop termination with notification through message passing.

We take two approaches to the problem of showing the soundness of our proof system. First, in the manner of Hoare and Lauer [HL74], we show in Section 4.3.1 that each

axiom and proof rule is consistent with respect to the operational semantics. Then, in the style of Owicki [Owi75, Owi76], in Section 4.3.2 we show the soundness of program annotations rather than of individual proof rules. This may seem like overkill, since the first proof should satisfy us that what the system proves is indeed true also for the operational semantics. We take both approaches because, while the first is simpler to follow, the second, though rather involved, introduces the notion of assertion functions to clarify what is required of a valid proof and to establish several results that we will use in Section 4.4 to show completeness.

We assume in our proofs in this chapter that we have some valid deductive system D for the natural numbers and any other data types and operations used in CSP programs. (D would, of course, not be effective.) We do not specify which deductive system is to be used. If L uses just the natural numbers, addition and multiplication, for example, it could be based on Peano's axioms. If we can derive q from p in D, we write $p \vdash q$.

4.3.1 Consistency of the Axioms and Proof Rules

Theorem 4.1 (Consistency) *The definition of the CSP fragment given by our axioms and rules is consistent with that given in the operational semantics.*

Proof: We prove that the implication associated with each rule and axiom by our definition of the deductive primitive in terms of the operational, is a valid derived theorem of the operational theory.

1. The assignment axiom is valid.

   Proof: We must show the validity of the implication associated with $\{p_e^x\}S_i\{p\}$:

$$\begin{aligned}\forall\sigma: \quad & (p_e^x(\sigma)\wedge \\ & <S_1 \parallel \ldots \parallel x := e \parallel \ldots \parallel S_N, \sigma> \rightarrow_1^\varepsilon \\ & \quad <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \mathcal{M}(x := e)(\sigma)>) \\ & \vdash p(\mathcal{M}(x := e)(\sigma))\end{aligned}$$

- Let the variables of S be $x, y_1, \ldots, y_n, n \geq 0$.
- Then $\sigma = (\sigma(x), \sigma(y_1), \ldots, \sigma(y_n))$.
- If, without loss of generality, the free variables of p are $x, y_1, \ldots, y_m$, where $0 \leq m \leq n$, then

$$p_e^x(\sigma) = p(\sigma_e^x) = p((\sigma(x), \sigma(y_1), \ldots, \sigma(y_m))_e^x) = p(\sigma(e), \sigma(y_1), \ldots, \sigma(y_m)).$$

- But, by the meaning of assignment,

$$p(\mathcal{M}(x := e)(\sigma)) = p(\sigma(e), \sigma(y_1), \ldots, \sigma(y_m)) = p(\sigma_e^x).$$

- This means that, in fact,

$$\begin{aligned}\forall\sigma: \quad & (<S_1 \parallel \ldots \parallel x := e \parallel \ldots \parallel S_N, \sigma> \rightarrow_1^\varepsilon \\ & \quad <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \mathcal{M}(x := e)(\sigma)>) \\ & \vdash (p(\sigma_e^x) \Leftrightarrow p(\mathcal{M}(x := e)(\sigma))),\end{aligned}$$

which is a stronger theorem than we need, so the assignment axiom is valid.

2. The preservation axiom is valid.

Proof: We must show the validity of the implication associated with $\{p\}S_i\{p\}$:

$$\begin{aligned}\forall\sigma, \tau, \quad & k \geq 0, h: \\ & (p(\sigma)\wedge \\ & <S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\ & \quad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau>) \\ & \vdash p(\tau)\end{aligned}$$

provided no free variable of p is subject to change in S_i.

- Let the variables of S be $x_1, \ldots, x_m, y_1, \ldots, y_n, m, n \geq 0$.
- Then

$$\sigma = (\sigma(x_1), \ldots, \sigma(x_m), \sigma(y_1), \ldots, \sigma(y_n)),$$

and

$$\tau = (\tau(x_1), \ldots, \tau(x_m), \tau(y_1), \ldots, \tau(y_n)).$$

- If, without loss of generality, the free variables of p are $x_1, \ldots, x_m$, and S_i changes only $y_1, \ldots, y_n$, then

$$\forall i : 1 \leq i \leq m : \sigma(x_i) = \tau(x_i),$$

so $p(\sigma) \equiv p(\tau)$.

- This means that, if no free variable of p is subject to change in S_i,

$$\begin{aligned} \forall \sigma, k \geq 0, h: \quad & (<S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\ & <S_1 \parallel \ldots \parallel after(S_i, S), \parallel \ldots \parallel S_N, \tau>) \vdash (p(\sigma) \Leftrightarrow p(\tau)), \end{aligned}$$

which is a stronger theorem than we need, so the preservation axiom is valid.

3. The input and output axioms are valid.

Proof: We must show the validity of the implication associated with

$$\{p\}c_i\{true\}$$

, for c_i an input or output command:

$$\begin{aligned} \forall \sigma: \quad & (p(\sigma) \wedge \\ & <S_1 \parallel \ldots \parallel c_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_1^\epsilon <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \mathcal{M}(c_i)(\sigma)>) \\ & \vdash true(\mathcal{M}(c_i)(\sigma)) \end{aligned}$$

– The implication given is equivalent to the disjunction

$$\begin{aligned}\forall\sigma: \ \neg(p(\sigma)\wedge\\ &<S_1 \parallel \ldots \parallel c_i \parallel \ldots \parallel S_N,\sigma> \rightarrow_i^{\epsilon}\\ &\quad <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N,\mathcal{M}(c_i)(\sigma)>)\\ \vee\ &true(\mathcal{M}(c_i)(\sigma))\end{aligned}$$

which holds since $true(\sigma)$ holds for any state. Thus the input and output axioms are valid for isolated communication commands.

4. The sequence rule is valid.

Proof: We must show the validity of the implication associated with

$$\frac{\{p\}S_i\{q\},\{q\}S_i'\{r\}}{\{p\}S_i;S_i'\{r\}}$$

which is

$$\begin{aligned}(\ \forall\sigma,\sigma',k\geq 0,h: \quad &(p(\sigma)\wedge <S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N,\sigma> \rightarrow_h^k\\ &<S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N,\sigma'>)\vdash q(\sigma'))\\ \wedge\ \forall\sigma',\tau,k'\geq 0,h': \quad &(q(\sigma')\wedge <S_1 \parallel \ldots \parallel S_i' \parallel \ldots \parallel S_N,\sigma'> \rightarrow_{h'}^{k'}\\ &<S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N,\tau>)\vdash r(\tau))\ )\\ \vdash\ (\ \forall\sigma,\tau,k''\geq 0,h'': \quad &(p(\sigma)\wedge <S_1 \parallel \ldots \parallel S_i;S_i' \parallel \ldots \parallel S_N,\sigma> \rightarrow_{k''}^{h''}\\ &<S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N,\tau>)\vdash r(\tau))\ ).\end{aligned}$$

– Assume that the antecedent conjunction is true, and that $p(\sigma)$ is true.

– For some σ,σ',h,k, assume without loss of generality

$$\begin{aligned}&<S_1 \parallel \ldots \parallel S_i;S_i' \parallel \ldots \parallel S_N,\sigma> \rightarrow_k^h\\ &\quad <S_1 \parallel \ldots \parallel after(S_i,S_i);S_i' \parallel \ldots \parallel S_N,\sigma'>.\end{aligned}$$

– But $after(S_i,S_i)\equiv E$, so we know, from the first clause of the antecedent, that $q(\sigma')$ must be true.

– Now, for σ' and some τ,h',k', assume without loss of generality

$$\begin{aligned}&<S_1 \parallel \ldots \parallel S_i' \parallel \ldots \parallel S_N,\sigma'> \rightarrow_{k'}^{h'}\\ &\quad <S_1 \parallel \ldots \parallel after(S_i',S_i') \parallel \ldots \parallel S_N,\tau>.\end{aligned}$$

- But $after(S_i', S_i') \equiv E$, so we know, from the second clause of the antecedent, that $r(\tau)$ must be true.
- Thus, we have

$$(<S_1 \parallel \ldots \parallel S_i; S_i' \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k+k'}^{h \circ h'} <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash r(\tau),$$

  with $r(\tau)$ true.
- Letting $h'' = h \circ h'$ and $k'' = k + k'$, we have

$$(p(\sigma) \wedge <S_1 \parallel \ldots \parallel S_i; S_i' \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k''}^{h''} <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash r(\tau),$$

  so the given implication is a valid theorem of the operational semantics, and the sequence rule is valid.

5. The alternation rule is valid.

   Proof: We must show the validity of the implication associated with

$$\frac{(\forall j : \{p \wedge b_j\} S_i^j \{q\})}{\{p\} \textbf{if } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ fi} \{q\}}$$

   which is

$$\begin{aligned} (\ \forall j, \sigma, \tau, k_j \geq 0, h_j : \ & ((p \wedge b_j)(\sigma) \wedge \\ & <S_1 \parallel \ldots \parallel S_i^j \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k_i}^{h_i} \\ & <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash q(\tau)) \\ \vdash (\ \forall \sigma, \tau, k \geq 0, h : \ & p(\sigma) \wedge \\ & (<S_1 \parallel \ldots \parallel \textbf{if } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ fi} \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\ & <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash q(\tau)\ ). \end{aligned}$$

   - Assume that the antecedent is true.
   - Suppose that $\forall j : 1 \leq j \leq m : \models \neg b_j(\sigma)$. Then there are no h and k such that the **if** statement terminates properly, and the consequent is vacuously true.

– On the other hand, if we suppose $\exists l : 1 \leq l \leq m : \models (p \wedge b_l)(\sigma)$, then

$$<S_1 \parallel \ldots \parallel \textbf{if } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ fi} \parallel \ldots \parallel S_N, \sigma> \rightarrow_i^{\epsilon}$$
$$<S_1 \parallel \ldots \parallel S_i^l \parallel \ldots \parallel S_N, \sigma>.$$

– But, from the antecedent, we have

$$(p \wedge b_l)(\sigma) \wedge (\quad <S_1 \parallel \ldots \parallel S_i^l \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k_l}^{h_l}$$
$$<S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash q(\tau),$$

so $q(\tau)$ holds if the alternation statement terminates, and the implication holds.

– Therefore the implication is a valid theorem of the operational semantics, and the alternation rule is valid.

6. The repetition rule is valid.

Proof: We must show the validity of the implication associated with

$$\frac{(\forall j : \{p \wedge b_j\} S_i^j \{p\})}{\{p\} \textbf{do } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ od} \{p \wedge (\forall j : \neg b_j)\}}$$

which is

$$\begin{aligned}
(\forall j, \sigma, \tau, k_j \geq 0, h_j : \quad & ((p \wedge b_j)(\sigma) \\
& \wedge <S_1 \parallel \ldots \parallel S_i^j \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k_j}^{h_j} \\
& \qquad <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \\
& \vdash p(\tau)) \\
\vdash (\forall \sigma, \tau, k \geq 0, h : \quad & p(\sigma) \\
& \wedge (<S_1 \parallel \ldots \parallel \textbf{do } []_{j=1:m} b_i \rightarrow S_i^j \textbf{ od} \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
& \qquad <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \\
& \vdash (p \wedge (\forall i : \neg b_i))(\tau)).
\end{aligned}$$

– Assume that the antecedent is true.

- Case 1: Suppose that $p(\sigma)$ is true and that $\forall l : 1 \leq l \leq m : \models \neg b_l(\sigma)$. Then

$$<S_1 \parallel \ldots \parallel \textbf{do } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ od} \parallel \ldots \parallel S_N, \sigma> \rightarrow_i^{\epsilon}$$
$$<S_1 \parallel \ldots \parallel after(\textbf{do } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ od}, \pi_i) \parallel \ldots \parallel S_N, \sigma>$$

  $\tau = \sigma$, and $(p \wedge (\forall j : \neg b_j))(\tau)$ is true, and the implication holds.

- Case 2: On the other hand, if we suppose $\exists l : 1 \leq l \leq m : \models (p \wedge b_l)(\sigma)$, then

$$<S_1 \parallel \ldots \parallel \textbf{do } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ od} \parallel \ldots \parallel S_N, \sigma> \rightarrow_i^{\epsilon}$$
$$<S_1 \parallel \ldots \parallel S_i^l; \textbf{do } []_{j=1:m} b_j \rightarrow S_i^j \textbf{ od} \parallel \ldots \parallel S_N, \sigma>$$

  But, from the antecedent, we have

$$(p \wedge b_l)(\sigma) \wedge (<S_1 \parallel \ldots \parallel S_i^l \parallel \ldots \parallel S_N, \sigma> \rightarrow_{k_l}^{h_l}$$
$$<S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>) \vdash p(\tau),$$

  so $p(\tau)$ holds if S_l terminates, and we again face the two cases with τ taking the place of σ.

- Since the **do** statement can terminate only if no b_i is true when the guards are evaluated, a terminating iteration always fits case 1.
- Therefore the implication is a valid theorem of the operational semantics, and the repetition rule is valid.

7. The consequence rule is valid.

   Proof: We must show the validity of the implication associated with

$$\frac{p \vdash p_1, \{p_1\} S_i \{q_1\}, q_1 \vdash q}{\{p\} S_i \{q\}}$$

which is

$$
\begin{aligned}
&(\forall \sigma, \tau, k \geq 0, h : \quad (p_1(\sigma) \wedge \\
&\qquad <S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
&\qquad\quad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau>) \vdash q_1(\tau) \\
&\qquad \wedge p \vdash p_1 \wedge q_1 \vdash q \;) \\
&\vdash (\forall \sigma, \tau, k \geq 0, h : \quad (p(\sigma) \wedge \\
&\qquad <S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
&\qquad\quad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau>) \vdash q(\tau) \;).
\end{aligned}
$$

- Assume that the antecedent is true, and that for some σ, τ, k, h, it is true that

$$
\begin{aligned}
&p(\sigma) \wedge \\
&<S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
&\quad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau>.
\end{aligned}
$$

- Since $p \vdash p_1$, this means that

$$
\begin{aligned}
&p_1(\sigma) \wedge \\
&<S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
&\quad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau> \vdash q(\tau).
\end{aligned}
$$

- But, since $q \vdash q_1$, this means that

$$
\begin{aligned}
&p_1(\sigma) \wedge \\
&\quad <S_1 \parallel \ldots \parallel S_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h \\
&\qquad <S_1 \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots \parallel S_N, \tau> \\
&\vdash q_1(\tau).
\end{aligned}
$$

- Therefore the implication is a valid theorem of the operational semantics, and the consequence rule is valid.

8. The satisfaction rule is valid.

Proof: We must show the validity of the implication associated with

$$\frac{\begin{array}{l}\{p\}c_i\{true\},\ p \vdash T_i = \vec{X},\\ \forall c_j : c_i \text{ matches } c_j \wedge \{q\}c_j\{r\} \wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i)) :\\ \qquad [(p \wedge q \wedge glue(c_i, c_j)) \vdash (s \wedge glue(c_i, c_j))^{x,T_i,T_j}_{e,T_i',T_j'}]\end{array}}{\{p\}c_i\{s\}}$$

where $T_i' = T_j' = synch(i, j, T_i, T_j)$. This is

$p \vdash T_i = \vec{X}$

$\wedge(\forall \sigma, \tau, k \geq 0, h :$

$p(\sigma)\wedge$

$(<S_1 \parallel \ldots \parallel c_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>)$

$\vdash true(\tau))$

$\wedge \forall c_j : c_i$ matches c_j

$\wedge \forall \sigma, \tau, k \geq 0, h :$

$[q(\sigma)$

$\wedge(<S_1 \parallel \ldots \parallel c_j \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>)$

$\vdash r(\tau)]$

$\wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i)) :$

$[(p \wedge q \wedge glue(c_i, c_j)) \vdash (s \wedge glue(c_i, c_j))^{x,T_i,T_j}_{e,T_i',T_j'}]$

$\vdash (\forall \sigma, \tau, k \geq 0, h[p(\sigma)\wedge$

$(<S_1 \parallel \ldots \parallel c_i \parallel \ldots \parallel S_N, \sigma> \rightarrow_k^h <S_1 \parallel \ldots \parallel E \parallel \ldots \parallel S_N, \tau>)$

$\vdash s(\tau_i)]$

- Assume that the antecedent conjunction holds, that $p(\sigma)$ and $q(\sigma)$ are true, and that c_i is a send $\pi_j!e$ which matches c_j, a receive $\pi_i?x$.
- If $p \vdash T_i = \vec{X} \wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{X}_j > \vec{Y}_j \vee \vec{Y}_i > \vec{X}_i))$ we have $c_i \parallel c_j$, so

$$<c_i||c_j, \sigma> \rightarrow_1^{(e,i,j)} <E, \mathcal{M}(x, T_i, T_j := e, T_i', T_j')(\sigma)>,$$

and thus

$$\begin{aligned}\tau &= \mathcal{M}(x, T_i, T_j := e, T_i', T_j')(\sigma) \\ &= (\sigma)^{x,T_i,T_j}_{e,T_i',T_j'}\end{aligned}$$

- Now, if $p(\sigma)$, $q(\sigma)$ and $glue(c_i, c_j)$ hold, then $(p \wedge q \wedge glue(s_i, r_j))(\sigma)$ is true, which implies that

$$(s \wedge glue(c_i, c_j))^{x,T_i,T_j}_{e,T_i',T_j'}(\sigma)$$

holds, which means that

$$(s \wedge glue(c_i, c_j))(\sigma)^{x,T_i,T_j}_{e,T_i',T_j'}$$

is true, which implies that

$$(s \wedge glue(c_i, c_j))(\tau)$$

holds. This implies that $s(\tau_j)$ holds.

- If we assume instead that c_i is a receive and c_j is a communicating send, then

$$<c_i||c_j, \sigma> \rightarrow_1^{(e,j,i)} <E, \mathcal{M}(x, T_i, T_j := e, T_i', T_j')(\sigma)>,$$

and the same line of reasoning applies.

- Therefore the implication associated with the satisfaction rule is a theorem of the operational semantics, and the rule is valid.

Thus all axioms and proof rules are consistent with the operational semantics.

∎

4.3.2 Assertion Functions and Soundness of an Annotation

We now show the soundness of an annotation of a CSP program devised using our proof system. We will show that if the pre- and post-conditions of the annotation satisfy the

requirements of the definition of *assertion functions*, then the annotation is sound. Assertion functions map assertions to program statements such that we can define both the pre– and postconditions of each statement and the causal relationships which must hold between all potentially communicating pairs of sends and receives.

First, in Definition 4.1, we define the assertion functions *pre*, *post* and *glue*.

Definition 4.1 Assertion functions: *Let pre and post be functions which map statements of S to assertions, and let glue be a function which maps pairs of communicating input and output statements to glue assertions. Then pre, post, and glue are assertion functions for $\{p\}S\{q\}$, where $p = \bigwedge_{i=1}^{N} p_i$ and $q = \bigwedge_{i=1}^{N} q_i$, if and only if they conform to the following constraints for each statement S_i^l of $S \equiv S_1 \parallel \ldots \parallel S_N$.*

1. for $i = 1 : N$, $p_i \vdash pre(S_i)$ and $post(S_i) \vdash q_i$;
2. if S_i^l is $x := e$, then $pre(S_i^l) \vdash post(S_i^l)_e^x$;
3. if S_i^l is *skip*, then $pre(S_i^l) \vdash post(S_i^l)$;
4. if S_i^l is $S_i^1; \ldots; S_i^m$, then
   (a) $pre(S_i^l) \vdash pre(S_i^1)$ and $post(S_i^m) \vdash post(S_i^l)$;
   (b) for $k = 1 : m-1$, $post(S_i^k) \vdash pre(S_i^{k+1})$;
5. if S_i^l is **if** $[]_{k=1:m} b_k \rightarrow S_i^k$ **fi**, then for $k = 1 : m$
   (a) $(pre(S_i^l) \wedge b_k) \vdash (pre(S_i^k))$;
   (b) $post(S_i^k) \vdash post(S_i^l)$;
6. if S_i^l is **do** $[]_{k=1:m} b_k \rightarrow S_i^k$ **od**, then for $k = 1 : m$
   (a) $(pre(S_i^l) \wedge b_k) \vdash (pre(S_i^k))$;
   (b) $post(S_i^k) \vdash pre(S_i^l)$;

(c) $(pre(S_i') \wedge \bigwedge_{k=1}^{m} \neg b_k) \vdash post(S_i')$;

7. if S_i' is a communication command, then for all matching communication commands S_j' such that $S_i' \parallel S_j'$,

   (a) Only the variables and vector clocks defined in π_i and π_j and the statement labels of S_i, S_j, π_i and π_j are free in $glue(S_i', S_j')$;

   (b) if $pre(S_i') \vdash T_i = \vec{X}$ and $pre(S_j') \vdash T_j = \vec{Y}$, then

$$\neg((pre \wedge post \wedge glue) \vdash (X_j > Y_j \vee Y_i > X_i));$$

   (c) $(pre(S_i') \wedge pre(S_j') \wedge glue(S_i', S_j'))$
$$\vdash (post(S_i') \wedge glue(S_i', S_j'))^{x,T_i,T_j}_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}.$$

Having given the definition of the assertion functions, we can show the relationships between assertion functions and proofs. In Theorem 4.2, we show that if we have assertions functions for a CSP program S with precondition p and postcondition q, then we can prove pre– and postconditions for each statement of S. As a corollary, we show that, given the same assumptions, we can use our proof system to prove $\{p\}S\{q\}$, that is, that if S starts execution in a state satisfying p, and terminates, it will terminate in a state satisfying q. In Theorem 4.4, we go in the other direction and prove that if there is a proof of $\{p\}S\{q\}$, then we can give assertion functions for $\{p\}S\{q\}$.

Theorem 4.2 *If pre, post, and glue are assertion functions for $\{p\}S\{q\}$, then it is possible to prove $\{pre(S_i')\}S_i'\{post(S_i')\}$ for each statement S_i' of S.*

Proof: Owicki [Owi75, Theorem 2.3] shows by induction on the structure of S_i' that the theorem is true for all but input and output commands. We extend the proof to include communication commands.

If S_i' is $\pi_j!e$ (or $\pi_j?x$), then by the input (respectively output) axiom, we have $\{pre(S_i')\}S_i'\{true\}$. Consider (the finite set of) all S_j' such that $S_i' \parallel S_j'$, where S_i' and S_j' are potentially communicating input and output commands. By the appropriate communication axiom, for any such S_j', we have $\{pre(S_j')\}S_j'\{true\}$. We see from Definition 4.1, and in particular from item (7) of that definition, that assertions $pre(S_i')$, $pre(S_j')$ and $glue(S_i', S_j')$ fulfill the requirements given in the antecedent of the satisfaction rule. This means that we can use the satisfaction rule to derive the postcondition of S_i' and show $\{pre(S_i')\}S_i'\{post(S_i')\}$.

Thus the theorem is true for all commands in our fragment of CSP.

∎

Corollary 4.3 *If pre, post, and glue are assertion functions for* $\{p\}S\{q\}$*, then it is possible to prove* $\{p\}S\{q\}$.

Proof: In the definition of the assertion functions, requirement (1) assures that for $i = 1 : N$, $p_i \vdash pre(S_i)$ and $post(S_i) \vdash q_i$. For each i, if S_i is empty, then by requirement (3) $pre(S_i) \vdash post(S_i)$, while if S_i is not empty, we can construct a proof outline, by the previous theorem, to show $\{pre(S_i)\}S_i\{post(S_i)\}$. Therefore we can show $\{p\}S\{q\}$.

∎

Theorem 4.4 *If there is a proof of* $\{p\}S\{q\}$*, then there are assertion functions for* $\{p\}S\{q\}$.

Proof: (Similar to the proof of Owicki's theorem 2.4 in [Owi75].) In a formal proof of $\{p\}S\{q\}$ (i.e., a proof in the same form as the proof of program POWER in Section 4.1), more than one line may refer to statement S_i' of process π_i. We eliminate

any line which does not contribute to the proof. From the remaining lines, there will be, for each S_i' where S_i' is neither an input nor an output command, one line which refers to S_i' and uses one of the axioms or proof rules. We use this line to define $pre(S_i')$ and $post(S_i')$. We justify the claims of any other lines which refer to S_i' by application of the consequence rule.

If S_i' is a communication command, then one line in the proof will establish

$$\{pre(S_i')\}S_i'\{true\}.$$

For each communication command S_j' such that $S_i' \parallel S_j'$, another will establish

$$\{pre(S_j')\}S_j'\{true\}.$$

Then, we must derive $post(S_i')$ in the proof by application of the satisfaction rule. Use of this rule satisfies requirement (7).

Owicki shows that the requirements are satisfied in the other cases, so all our pre- and post-conditions satisfy the requirements of the assertion functions.

■

Now that we have shown the close relationship between a proof and assertion functions, we can show that the preconditions and postconditions given by the assertion functions correctly describe program state during any program execution. We want to prove that if a given statement R_i of S is executed during any run of S, then the assertions mapped to R_i by the assertion functions accurately describe its precondition and postcondition. We take two steps to prove this. First, in Theorem 4.5, we handle a relatively simple case. We consider a statement A_i, where A_i is one of the statements covered by the

first case in the definition of the *after* relation in Section 4.3. Recall that these are the statements that are not contained within some guarded command or within a statement in a sequence of statements. Suppose that the assertion functions accurately define the postcondition of A_i in the execution of S. We show that, for a statement $S_i \equiv after(A_i, \pi_i)$, if $S_i \equiv after(R_i, \pi_i)$, the assertion functions accurately define the postcondition of R_i. If, on the other hand, $S_i \equiv before(R_i, \pi_i)$, the assertion functions accurately define the precondition of R_i. We then use Theorem 4.5 in Theorem 4.6 to prove that the assertion functions validly define the pre– and post–conditions of an arbitrary statement R_i in S.

Theorem 4.5 *Suppose that pre, post and glue are assertion functions for* $\{p\}S\{q\}$, A_i *is an input, output, assignment, skip or do statement in* S, *and* R_i *is a statement in* S. *If* $<S_1||\cdots||S_N, \sigma> \rightarrow_k^h <S_1'||\cdots||S_N', \tau>$ *for some* $h, k, S_1', \ldots S_N'$, $S_i' = after(A_i, \pi_i)$ *and* $\models post(A_i)(\tau)$, *then*

1. *if* $S_i' = after(R_i, \pi_i)$ *then* $\models post(R_i)(\tau)$;
2. *if* $S_i' = before(R_i, \pi_i)$ *then* $\models pre(R_i)(\tau)$;

Proof of claim 1: By induction on the structure of R_i. If R_i is an assignment, input, output, skip or do statement, then $R_i = A_i$ and claim 1 is given.

If R_i is $R_i^1, \ldots, R_i^m$, then S_i' must be $after(R_i^m, \pi_i)$, and by induction $\models post(R_i^m)(\tau)$. Since, by definition of the assertion functions, $post(R_i^m) \vdash post(R_i)$, $\models post(R_i)(\tau)$.

If R_i is **if** $[]_{k=1:m} b_k \rightarrow R_i^k$ **fi**, $S_i' = after(R_i^k, \pi_i)$ for some $j = 1 : m$. For any such j, $\models post(R_i^m)(\tau)$, and since $post(R_i^m) \vdash post(R_i)$, $\models post(R_i)(\tau)$.

Proof of claim 2: We are given that $after(A_i, \pi_i) \equiv S_i' \equiv before(R_i, \pi_i)$, so by definition of the *before* relation, $S_i' \equiv R_i; after(R_i, \pi_i)$. Now, by the definition of the

after relation, we see that either 1) R_i follows A_i in a sequence of statements, 2) that R_i is a **do** statement in which A_i is the last statement of one of the guarded commands, or 3) that R_i follows an **if** statement A_i' in a sequence of statements, where A_i is the last statement of one of the guarded commands of A_i'. We consider the three cases.

1. By requirement 4(a) of the definition of the assertion function, $post(A_i) \vdash pre(R_i)$, so $\models pre(R_i)$.
2. If A_i' is the guarded command of which A_i is the last statement, we know by the proof of the first claim that since $\models post(A_i)$, $\models post(A_i')$. Since $post(A_i') \vdash pre(R_i)$ by the definition of the assertion functions, we have $\models pre(R_i)$.
3. Similarly, by the proof of the first claim we know that since $\models post(A_i)$, $\models post(A_i')$. Since R_i follows A_i' in a sequence of statements, $post(A_i') \vdash pre(R_i)$, and so we have $\models pre(R_i)$.

∎

Theorem 4.6 *If pre, post and glue are assertion functions for $\{p\}S\{q\}$, R_i is a statement in S, and $<S_1||\cdots||S_N,\sigma> \rightarrow_k^h <S_1'||\cdots||S_N',\tau>$ for some $h, k, S_1', \ldots S_N'$, with $\models p(\sigma)$, then*

1. *if $S_i' = before(R_i, \pi_i)$ then $\models pre(R_i)(\tau)$;*
2. *if $S_i' = after(R_i, \pi_i)$ then $\models post(R_i)(\tau)$.*

Proof: We use induction on k.

If $k = 0$, claim 1 is true because $\models p(\sigma)$, $p \vdash pre(R_i)$, and so $\models pre(R_i)(\sigma)$. Claim 2 is vacuously true.

Suppose $k > 0$. Then

$$<S_1||\cdots||S_N,\sigma> \rightarrow_{k-1}^{h_1} <S_1''||\cdots||S_N'',\sigma'>$$

and

$$<S_1''||\cdots||S_N'',\sigma'> \rightarrow_1^{h_2} <S_1'||\cdots||S_N',\tau>,$$

where $h = h_1 \circ h_2$.

1. Suppose that the kth step was a local step in π_j, $i \neq j$. Then since $k-1<k$, by induction we see that if $S_i'' = \mathit{before}(R_i, \pi_i)$, then $\models \mathit{pre}(R_i)(\sigma')$. Since $\sigma_i' = \tau_i$, this means that $\models \mathit{pre}(R_i)(\tau)$. Similarly, if $S_i'' = \mathit{after}(R_i, \pi_i)$, then $\models \mathit{post}(R_i)(\sigma')$. and so $\models \mathit{post}(R_i)(\tau)$.

2. Suppose that the kth step was a local step in π_i. Consider the cases for this step, the execution of A_i:

   (a) A_i is the assignment $x := e$. Thus $S_i' = \mathit{after}(A_i, \pi_i)$. By induction, $\models \mathit{pre}(A_i)(\sigma')$. Since $\mathit{pre}(A_i) \vdash \mathit{post}(A_i)_e^x$ by Definition 5.1, we have $\models \mathit{post}(A_i)_e^x(\sigma')$, which implies $\models \mathit{post}(A_i)(\sigma')_e^x$. But, $\tau \in \mathcal{M}(x := e)(\sigma')$ and $(\sigma')_e^x \in \mathcal{M}(x := e)(\sigma')$, so $\models \mathit{post}(A_i)(\sigma')_e^x$ implies that $\models \mathit{post}(A_i)(\tau)$. By the previous theorem, claims 1 and 2 are satisfied.

   (b) A_i is a **skip**. Thus $S_i' = \mathit{after}(A_i, \pi_i)$. By induction, $\models \mathit{pre}(A_i)(\sigma')$. Since $\mathit{pre}(A_i) \vdash \mathit{post}(A_i)$, we have $\models \mathit{post}(A_i)(\sigma')$, But, a **skip** changes no variable values, so $\sigma' = \tau$, so $\models \mathit{post}(A_i)(\tau)$. Applying the previous theorem, claims 1 and 2 are satisfied.

   (c) A_i is **if** $[]_{k=1:m} b_k \rightarrow A_i^k$ **fi**.

   By induction, $\models \mathit{pre}(A_i)(\sigma')$. If, for some ℓ, $\sigma'(\ell)$ is true, then $S_i' = \mathit{before}(A_i^\ell, \pi_i)$, and $\models (\mathit{pre}(A_i) \wedge b_\ell)(\sigma')$.

   But, $(\mathit{pre}(A_i) \wedge b_\ell) \vdash \mathit{pre}(A_i^\ell)$, and by the definition of the operational semantics of the **if** statement, $\sigma' = \tau$, so $\models \mathit{pre}(A_i^\ell)(\tau)$.

But if $S_i' = before(R_i, \pi_i)$, then $before(R_i, \pi_i) = before(A_i^\ell, \pi_i)$, and so $\models pre(R_i)(\tau)$. Thus claim 1 is satisfied, while claim 2 is vacuously true.

(d) A_i is **do** $[]_{k=1:m} b_k \rightarrow A_i^k$ **od**. If, for some ℓ, $\sigma'(b_\ell)$ is true, then since $\models pre(A_i)(\sigma')$ by induction, $(pre(A_i) \wedge b_\ell) \vdash pre(A_i^\ell)$, and $\sigma' = \tau$, $\models pre(A_i^\ell)(\tau)$. Also, $S_i' = before(R_i, \pi_i) = before(A_i^\ell, \pi_i)$, so $\models pre(R_i)(\tau)$. Thus claim 1 is satisfied, while claim 2 is vacuously true.

If, on the other hand, for no ℓ is $\sigma'(b_\ell)$ true, then $\sigma' = \tau$ and $S_i' = after(A_i, \pi_i)$. Since $\models pre(A_i)(\sigma')$ by induction, and $(pre(A_i) \wedge \bigwedge_{k=1}^m \neg b_k) \vdash post(S_i)$, we have $\models pre(A_i)(\tau)$. By the previous theorem, claims 1 and 2 are satisfied.

3. Suppose the kth step is a communication from π_i to π_j. Let A_i be s_i and let A_j be r_j, where $s_i \parallel r_j$. Then $S_i' = after(A_i, \pi_i)$, and $S_j' = after(A_j, \pi_i)$. By induction, $\models pre(A_i)(\sigma')$ and $\models pre(A_j)(\sigma')$, so $\models (pre(A_i) \wedge pre(A_j) \wedge glue(A_i, A_j))(\sigma')$, which by the definition of the assertion functions implies

$$\models (post(A_i) \wedge post(A_j) \wedge glue(A_i, A_j))^{x,T_i,T_j}_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}(\sigma'),$$

which implies

$$\models (post(A_i) \wedge post(A_j) \wedge glue(A_i, A_j))(\sigma')^{x,T_i,T_j}_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}.$$

But since

$$(\sigma')^{x,T_i,T_j}_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)} \in \mathcal{M}(s_i \parallel r_j)(\sigma') \text{ and } \tau \in \mathcal{M}(s_i \parallel r_j)(\sigma'),$$

we have

$$\models (post(A_i) \wedge post(A_j) \wedge glue(A_i, A_j))(\tau).$$

By the previous theorem, claims 1 and 2 are satisfied for both R_i and R_j.

∎

Now that we have shown the relationship between proofs and assertion functions, and proven that assertion functions define valid statement pre– and post–conditions, we can show that our proof system is sound. It is easy to prove Theorem 4.7, which states that what we prove with our proof system is, in fact, true. We simply note that if we can prove $\{p\}S\{q\}$, then there are assertion functions for $\{p\}S\{q\}$, and then use Theorem 4.6 to show that this means that the proof is valid. This demonstrates the soundness of annotations derived by our proof system, and completes our proof of soundness.

Theorem 4.7 (Soundness of annotation) *If* $\vdash \{p\}S\{q\}$, *then* $\models \{p\}S\{q\}$,

Proof: By Theorem 4.4, there exist assertion functions for $\{p\}S\{q\}$. Suppose

$$<S_1||\cdots||S_N,\sigma> \rightarrow_k^h <S_1'||\cdots||S_N',\tau>$$

for some $h, k, S_1', \ldots S_N'$, where $\forall i : \ S_i' = E$, and $\models p(\sigma)$. Then by the previous theorem $\models post(S_i)(\tau)$, and since $post(S_i) \vdash q_i$, $\models q_i(\tau)$, and so $\models \{p\}S\{q\}$.

4.4 Relative Completeness of the Proof System

No axiomatic proof system is complete for a programming language which includes the natural numbers, in part since there is no complete first-order deductive system for the natural numbers. We get around this difficulty by assuming that we have such a deductive system, and then showing that if we use it to make deductions in our proof rules, our system can prove whatever is true. This technique, suggested by Cook [Coo78], was also used by both Apt [Apt83] and Owicki [Owi75, Owi76] to prove the relative completeness of their proof systems.

Both Apt and Owicki use auxiliary variables in their completeness proofs to capture an encoding of the initial state of each process and of the history of interprocess communications. Apt has shown [ABM79, Apt81] that only the use of auxiliary variables in this way can make a system which uses only recursive assertions (rather than recursively

enumerable assertions) complete. He showed, for instance, that for a proof system which uses only control predicates, recursive assertions are not powerful enough.

We do not use auxiliary variables, but we accept the force of Apt's argument that we must be able to record the initial state and history of a computation to achieve completeness with recursive assertions. We therefore use process vector clocks in our completeness proof as Owicki and Apt used auxiliary variables in their proofs.

In this section, we define how we encode initial state and history using vector clocks. Then, in Section 4.4.1, we define assertion functions for an arbitrary CSP program, and show that these functions are recursive. We prove in Section 4.4.2 that these generic functions satisfy the requirements of Definition 4.1, which defined assertion functions. This result lets us show in Section 4.4.3 that our proof system is relatively complete.

To record initial state and history, any invertible encoding technique will do that represents sequences of natural numbers so that the sequence can be extended and recovered. We indicate the encoding of the sequence x_1, x_2, x_3 by $\ll x_1, x_2, x_3 \gg$, and extend it by concatenation, which we designate by $\circ$.

We initialize $T_{i,i}$ to encode the initial values of $\vec{x_i}$, the list of π_i's variables. Then, when we update the clocks of π_i and π_j at the exit point of a synchronous send from π_i to π_j, we set

$$T_{i,i} := T_{i,i} \circ \ll e, i, j \gg$$

$$. \; T_{j,j} := T_{j,j} \circ \ll x, i, j \gg$$

before we take the *sup* of T_i and T_j. Since this concatenation will have the effect of adding some positive value to $T_{i,i}$ and $T_{j,j}$, this update policy satisfies the requirements of the definition given in Section 2.2.4, where we said that $T_{i,i}$ and $T_{j,j}$ should be incremented by some value $d > 0$.

4.4.1 Functions *pre*, *post*, and *glue*.

We want first to define generic assertion functions for an arbitrary CSP program. To this end, let $\sigma = \bigcup_{i=1:N} \sigma_i$ be a global state such that $<S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1' \parallel \ldots \parallel S_N', \sigma>$, for some h, k, program S and initial state τ.

To be able to achieve relative completeness, we need to be able to describe the initial states of all processes which have communicated with π_i, and all the interprocess communications which can causally affect σ_i, in the execution whose history is recorded by h and encoded in process state σ. Because we initialize vector time by encoding process state and update it by encoding records of communication, we can use process vector clocks to do this. Recall that h is a sequence of records of communication. Let h record the history of a computation defined by the sequence $[<S^0, \sigma^0>], [<S^1, \sigma^1>], \ldots, [<S^m, \sigma^m>]$, where S^i is $S_1^i \parallel \ldots \parallel S_N^i$. Also, let vector clocks be initialized and updated as explained above. Then we can define $past(\pi_i, h)$ to be the N element vector $[t_1, \ldots, t_N]$, where the jth element in $past(\pi_i, h)$ is the value of $T_{j,j}$ at the point at which π_j last communicated with π_i in the computation recorded by h. More formally, we can say

$$\forall j, k : 1 \leq j \leq N, 0 \leq k \leq m : t_j = \max\{\sigma_j^k(T_{j,j}) | \sigma_j^k(T_{j,j}) \leq \sigma_i^m(T_{i,j})\}.$$

Of course, this is not just some arbitrary vector. Define h_i to be the sequence of records of communication obtained from h by extracting, in order, all records of sends or receives in which π_i participated. These are the records (a, i, j) and (a, j, i). Vector clock element $T_{i,i}$ encodes the initial state of π_i and h_i. Moreover, when π_i communicates with process π_j, the clock update function *synch* ensures that element $T_{i,j}$ encodes both the initial state of π_j and h_j, the records of the communications in which π_j has participated, up to and including the current communication. This means that when a computation has reached state σ with history h, each vector clock $T_i = past(\pi_i, h)$.

With this in hand, we can give definitions of recursive functions which satisfy the requirements for assertion functions given in Definition 4.1. We need our assertion functions

to be recursive to guarantee that any assertion we use in an annotation is computable. We define

1. glue(s_i, r_j), which describes the states of π_i and π_j at the entry and exit points of semantically matching send $s_i : \pi_j!e$ and receive $r_j : \pi_i?x$, as:

$$\begin{aligned}\models glue(s_i, r_j)(\sigma) \equiv \exists \sigma', \tau, S'_1, \ldots, S'_N, k, h[\ & \models p(\tau),\\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h\\ & \qquad <S'_1 \parallel \ldots \parallel S'_N, \sigma'>,\\ & [(S'_i \equiv before(s_i, \pi_i) \wedge \sigma'_i = \sigma_i]\\ & \quad \vdash [S'_j \equiv before(r_j, \pi_j)) \vdash \sigma'_j = \sigma_j],\\ & [(S'_i \equiv after(s_i, \pi_i) \wedge \sigma'_i = \sigma_i]\\ & \quad \vdash [S'_j \equiv after(r_j, \pi_j)) \vdash \sigma'_j = \sigma_j],\\ & \sigma_i(T_i) = past(\pi_i, h),\\ & \sigma_j(T_j) = past(\pi_j, h)].\end{aligned}$$

2. pre(R_i), which describes the state of π_i at the entry point of statement R_i of π_i:

$$\begin{aligned}\models pre(R_i)(\sigma) \equiv \exists \sigma', \tau, S'_1, \ldots, S'_N, k, h[\ & \models p(\tau),\\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h\\ & \qquad <S'_1 \parallel \ldots \parallel S'_N, \sigma'>,\\ & \sigma'_i = \sigma_i,\\ & S'_i \equiv before(R_i, \pi_i),\\ & \sigma_i(T_i) = past(\pi_i, h)].\end{aligned}$$

3. post(R_i), which describes the state of π_i at the exit point of statement R_i of π_i:

$$\begin{aligned} \models post(R_i)(\sigma) \equiv \exists \sigma', \tau, S'_1, \ldots, S'_N, k, h[\quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \qquad <S'_1 \parallel \ldots \parallel S'_N, \sigma'>, \\ & \sigma'_i = \sigma_i, \\ & S'_i \equiv after(R_i, \pi_i), \\ & \sigma_j(T_j) = past(\pi_j, h)]. \end{aligned}$$

These functions are recursive. We prove this using the technique used by Apt [Apt81] to show that similar assertions, but with auxiliary variables, for a proof system for Owicki's GPL language for parallel programs, are recursive. First, we define what it means for a program to be *derived* from another program.

Definition 4.2 *A program S' is* derived from *program S if and only if $S' \equiv S'_1 \parallel \ldots \parallel S'_N$ and for $i = 1, \ldots, N$, $S'_i \equiv before(R_i, \pi_i)$ or $S'_i \equiv after(R_i, \pi_i)$ for statement R_i of π_i.*

This tells us that S' is derived from S if and only if there exist some τ, σ, h and k such that $<S, \tau> \rightarrow_k^h <S', \sigma>$. Thus S' is what remains to be executed after a k step computation of S from initial state τ with history h.

Next, we show in Lemma 4.8 that we can use a recursive procedure to determine whether we can get from one given program/state configuration to another in one step with a given record of communication and encoding of that record. We use this result in Lemma 4.9 to show that we can use a recursive procedure to determine whether we can get from a given program/state configuration to another in k steps with a given history and encoding of that history. We can then use Lemma 4.9 in Theorem 4.10 to prove that $pre(R_i)(\sigma)$, $post(R_i)(\sigma)$, and $glue(s_i, r_j)$ are recursive.

Lemma 4.8 *The relation V defined by*

$$V(S', S'', \sigma, \tau, r, m) \equiv \begin{array}{l} S' \text{ is derived from } S; \\ <S', \tau> \rightarrow_1^r <S'', \sigma>; \\ r \text{ is a record of communication or } \epsilon; \\ r \text{ is encoded by } m \end{array}$$

is recursive.

Proof:

- We can determine whether S' is derived from S by checking each S_i'.
- We make the transition from τ to σ in one step. This step must involve either the evaluation of a boolean expression, the execution of an assignment or a skip statement, the execution of a **do** statement with no true guards, or the execution of a synchronous communication. Using the definitions of the $\rightarrow$, $\rightarrow_k^h$ and $\mathcal{M}$ relations, we can easily check whether $<S', \tau> \rightarrow_1^r <S'', \sigma>$ for a given S', S'', σ, τ, and r.
- We can also easily check whether r is ϵ or the appropriate record of communication (a, i, j).
- The relation "r is encoded by m" is also recursive given S', S'', σ, τ, and r such that $<S', \tau> \rightarrow_1^r <S'', \sigma>$. If the step executed is a local operation in one process, then $r = \epsilon$ and $m = \ll \gg$. If the step executed is a communication $s_i : \pi_j!e \parallel r_j : \pi_i?x$, then $r = (e, i, j)$ and $m = \ll e, i, j \gg$.

Thus V is recursive.

∎

Lemma 4.9 *The relation U defined by*

$$U(S', S'', \sigma, \tau, h, k, m) \equiv \begin{array}{l} S' \text{ is derived from } S; \\ <S', \tau> \rightarrow_k^h <S'', \sigma>; \\ h \text{ is encoded by } m \end{array}$$

is recursive.

Proof: We prove that U is recursive by giving a recursive redefinition of it which requires that only a recursive procedure be performed at each level of recursion. We redefine $U(S', S'', \sigma, \tau, h, k, m)$ as

1. S' is derived from S; and
2. Either
   (a) $h = \epsilon$, $S' \equiv S''$, $\sigma = \tau$ and $m = \ll \gg$; or
   (b) $h \neq \epsilon$, $<S', \tau> \rightarrow_1^r <S^0, \sigma^0>$, r is encoded by m_1, and

$$U(S^0, S'', \sigma, \sigma^0, h_1, k-1, m_2)$$

   , where
   i. $h = r \circ h_1$;
   ii. if $r = \epsilon$, then if the step executed was a boolean evaluation, a **do** with no true guards, or a **skip**, then $\sigma^0 = \tau$, else if the step executed was $x := e$, then $\sigma^0 = \tau_e^x$;
   but if $r = (a, i, j)$ and $s_i : \pi_j!a \parallel r_j : \pi_i?y$ was executed, then $\sigma^0 = \tau_a^y$;
   iii. if $r = (a, i, j)$, then S^0 must be

$$S_1' \parallel \ldots \parallel after(S_i', \pi_i) \parallel \ldots \parallel after(S_j', \pi_j) \parallel \ldots \parallel S_N'$$

   otherwise S^0 must be

$$S_1' \parallel \ldots \parallel after(S_i', \pi_i) \parallel \ldots \parallel S_N'$$

   for some $i : 1 \leq i \leq N$;

iv. either $m_1 = \ll \gg$ and $m_2 = m$, or

$m_1 = \ll a, i, j \gg$ and $m_2 = \ll r_1 \circ \ldots \circ r_l \gg$ where $m = \ll m_1 \circ r_1 \circ \ldots \circ r_l \gg$.

Since there are only a finite number of possibilities for S^0 and σ^0, given S' and r, we can determine $V(S', S^0, \sigma^0, \tau, r, m_1)$ for each possibility. By the previous lemma we know that we can check V recursively. We also know that we will recurse on U in our definition only a finite number of times, since k is finite. Therefore U is recursive.

■

Theorem 4.10 *The assertions $pre(R_i)(\sigma)$, $post(R_i)(\sigma)$ and $glue(s_i, r_j)(\sigma)$ are recursive.*

Proof: We will show the proof for $pre(R_i)(\sigma)$. The other two assertions are handled in the same way.

- We can compute the initial state τ from σ'. The first element encoded in each process vector clock T_{ii} encodes the initial values of the variables of π_i. Thus for some recursive function f, $\tau = f(\sigma')$.
- There are only finitely many programs derived from S, since each of its N constituent processes contains only a finite number of statements. Thus

$$\begin{aligned} \models pre(R_i)(\sigma) \equiv \bigvee_{S' \text{ derived from } S} \exists \sigma', h, k: \quad & p(f(\sigma')), \\ & <S, \tau> \rightarrow_k^h <S', \sigma'>; \\ & \sigma'_i = \sigma_i, \\ & S'_i \equiv before(R_i, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

- By the previous lemma, checking whether $pre(R_i)(\sigma)$ holds for any particular history is recursive.

- So, to show $pre(R_i)(\sigma)$ recursive, we need now only to be able to bound (the encodings of) all possible histories h such that $\sigma_i(T_i) = past(\pi_i, h)$, with a bound that depends only on σ'.
- Each history is of the form

$$(a_1, i_1, j_1) \circ \ldots \circ (a_k, i_k, j_k),$$

  which we encode as

$$\ll\ll a_1, i_1, j_1 \gg \circ \ldots \circ \ll a_k, i_k, j_k \gg\gg$$

- To bound any such history, we need to encode some k element sequence, each of whose elements is larger than any element of h. The problem is that a bound for the first element in each record of communication, the value transmitted, is not obvious. To give a bound, though, we can use one of the characteristics of vector clocks.

  As Mattern shows in [Mat89], in global state σ, any clock T_i is less than or equal in value to the vector $<T_{11}, T_{22}, \ldots, T_{nn}>$. This is a vector whose ith element, for all $i : 1 \ldots N$, is the ith element of T_i. Since no other process has more knowledge of π_i's state than does π_i, no clock element $T_{j,i}$ for $i \neq j$ will have a value greater than $T_{i,i}$. Putting together all these maximal elements yields a vector which is an upper bound on the clock values in σ.
- We can define an encoding $\mathcal{T}$ of the elements of such a maximal clock vector: $\mathcal{T} = \ll\sigma'(T_{1,1}), \sigma'(T_{2,2}), \ldots, \sigma'(T_{N,N})\gg$. Any encoded data value in a record of communication from h will be less in value than an encoding of $\mathcal{T}$, since the record itself must be encoded within the elements of $\mathcal{T}$ by the vector clock update scheme. Therefore the encoding of a sequence which consists of the tuple $(N, N, \mathcal{T})$, repeated k times, is greater in value than the encoding of any possible history h.

– So, having shown that we can place a bound on h that depends solely on σ', we have proved that $pre(R_i)(\sigma)$ is recursive.

4.4.2 *pre*, *post*, and *glue* as Assertion Functions

We must now show that these definitions satisfy the requirements for assertion functions given in Section 4.3.2. This will let us prove relative completeness, since we know that if there are assertion functions for a CSP program, we can write a proof of it using our axioms and proof rules.

Theorem 4.11 *The functions pre, post and glue are assertion functions for* $\{p\}S\{q\}$.

Proof: To show that the function definitions satisfy the requirements for assertion functions, we show that they satisfy the requirements of each of the 7 parts of Definition 4.1.

1. We must show that a) $p_i(\sigma) \vdash pre(S_i)(\sigma)$, and b) $post(S_i)(\sigma) \vdash q_i(\sigma)$.

   (a) By definition,

$$\begin{aligned} \models pre(S_i)(\sigma) \equiv \exists \sigma', \tau, S'_1, \ldots, S'_N, k, h : \; & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \qquad <S'_1 \parallel \ldots \parallel S'_N, \sigma'>, \\ & \sigma'_i = \sigma_i, \\ & S'_i \equiv before(R, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

   Letting $k = 0$ and $h = \epsilon$, we have $\sigma \equiv \tau$, so if $\models p_i(\sigma)$, then $\models pre(S_i)(\sigma)$, and therefore $p_i(\sigma) \vdash pre(S_i)(\sigma)$.

(b) We have

$$\models post(S_i)(\sigma) \equiv \exists \sigma', \tau, S_1', \ldots, S_N', k, h: \quad \models p(\tau),$$
$$<S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h$$
$$<S_1' \parallel \ldots \parallel S_N', \sigma'>,$$
$$\sigma_i' = \sigma_i,$$
$$S_i' \equiv after(R, \pi_i),$$
$$\sigma_j(T_j) = past(\pi_j, h).$$

Assume $\models \{p\}S\{q\}$. This means that $\forall \sigma, \tau, k, h$:

$$(p(\tau) \wedge$$
$$<S_1 \parallel \ldots \parallel S_i \parallel \ldots S_N, \tau> \rightarrow_k^h$$
$$<S_1' \parallel \ldots \parallel after(S_i, \pi_i) \parallel \ldots S_N', \sigma>)$$
$$\vdash q(\sigma).$$

Using these definitions, we see that $\models post(S_i)(\sigma)$ implies that $\models q(\sigma)$, and since $q \vdash q_i$, that $\models q_i(\sigma)$. Therefore $post(S_i)(\sigma) \vdash q_i(\sigma)$.

2. If S_i' is $x := e$, we must show $pre(S_i')(\sigma) \vdash pre(S_i')_e^x(\sigma)$.

   - Assume $\models pre(S_i')(\sigma)$. Then

$$\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad \models p(\tau),$$
$$<S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>,$$
$$\sigma_i' = \sigma_i,$$
$$S_i'' \equiv before(S_i', \pi_i),$$
$$\sigma_i(T_i) = past(\pi_i, h).$$

But $before(S_i', \pi_i) \equiv S_i'; after(S_i', \pi_i)$, and

$$<S_i'; after(S_i', \pi_i), \sigma'> \rightarrow <after(S_i', \pi_i), \mathcal{M}(x := e)(\sigma')>.$$

So for some ς

$$\begin{aligned}\models \exists \varsigma', \tau, S_1'', \ldots, S_N'', k', h' : \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_{k'}^{h'} \\ & \quad <S_1'' \parallel \ldots \parallel after(S_i', \pi_i) \parallel \ldots \parallel S_N'', \varsigma'>, \\ & \varsigma' = \mathcal{M}(x := e)(\sigma'), \\ & \varsigma_i' = \varsigma_i, \\ & k' = k + 1, h' = h, \\ & \sigma_i(T_i) = past(\pi_i, h').\end{aligned}$$

Thus $\models post(S_i')(\varsigma)$. But $\sigma_e'^x \in \mathcal{M}(x := e)(\sigma')$, so we have $\models post(S_i')(\sigma_e^x)$, which implies $\models post(S_i')_e^x(\sigma)$. Therefore $pre(S_i')(\sigma) \vdash post(S_i')_e^x(\sigma)$.

3. If S_i' is **skip**, we must show $pre(S_i')(\sigma) \vdash post(S_i')(\sigma)$.

   - Assume $\models pre(S_i')(\sigma)$. Then

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h : \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv before(S_i', \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

   But $before(S_i', \pi_i) \equiv S_i'; after(S_i', \pi_i)$, and

$$<S_i'; after(S_i', \pi_i), \sigma'> \rightarrow <after(S_i', \pi_i), \sigma'>,$$

   so

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k', h' : \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_{k'}^{h'} \\ & \quad <S_1'' \parallel \ldots \parallel after(S_i', \pi_i) \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & k' = k + 1, h' = h, \\ & \sigma_i(T_i) = past(\pi_i, h').\end{aligned}$$

Thus $\models post(S_i')(\sigma)$, so $pre(S_i')(\sigma) \vdash post(S_i')(\sigma)$.

4. If S_i' is $S_i^1; \ldots; S_i^m$, we must demonstrate that (a) $pre(S_i')(\sigma) \vdash pre(S_i^1)(\sigma)$, (b) $post(S_i^m)(\sigma) \vdash post(S_i')(\sigma)$ and (c) for $k = 1 : m-1, post(S_i^k)(\sigma) \vdash pre(S_i^{k+1})(\sigma)$.

   Therefore

   (a) Assume $\models pre(S_i')(\sigma)$. Then

$$\begin{aligned} \models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv before(S_i', \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

   But

$$\begin{aligned} before(S_i', \pi_i) &\equiv S_i'; after(S_i', \pi_i) \\ &\equiv S_i^1; \ldots; S_i^m; after(S_i', \pi_i) \\ &\equiv S_i^1; after(S_i^1, S_i); after(S_i', \pi_i) \\ &\equiv S_i^1; after(S_i^1, \pi_i) \\ &\equiv before(S_i^1, \pi_i) \end{aligned}$$

   so $\models pre(S_i^1)(\sigma)$, and therefore $pre(S_i')(\sigma) \vdash pre(S_i^1)(\sigma)$.

   (b) Assume $\models post(S_i^m)(\sigma)$. Then

$$\begin{aligned} \models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv after(S_i^m, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

But

$$\begin{aligned} after(S_i^m, \pi_i) &\equiv after(S_i^m, S_i'); after(S_i', \pi_i) \\ &\equiv E; after(S_i', \pi_i) \\ &\equiv after(S_i', \pi_i), \end{aligned}$$

so $\models post(S_i')(\sigma)$, and thus $post(S_i^m) \vdash post(S_i')(\sigma)$.

(c) Assume that for some k, $1 \leq k < m$, $\models post(S_i^k)(\sigma)$. Then

$$\begin{aligned} \models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \ & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv after(S_i^k, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

But

$$\begin{aligned} after(S_i^k, \pi_i) &\equiv after(S_i^k, S_i'); after(S_i', \pi_i) \\ &\equiv S_i^{k+1}; \ldots; S_i^m; after(S_i', \pi_i) \\ &\equiv S_i^{k+1}; after(S_i^{k+1}, \pi_i) \\ &\equiv before(S_i^{k+1}, \pi_i), \end{aligned}$$

so $\models pre(S_i^{k+1})(\sigma)$, and therefore, for $k = 1 : m-1$,

$$post(S_i^k)(\sigma) \vdash pre(S_i^{k+1})(\sigma).$$

5. If S_i' is **if** $[]_{k=1:m} b_k \rightarrow S_i^k$ **fi**, then we must show that for $k = 1 : m$,

(a) $(pre(S_i') \wedge b_k)(\sigma) \vdash (pre(S_i^k)(\sigma))$; and (b) $post(S_i^k)(\sigma) \vdash post(S_i')(\sigma)$.

(a) Assume $\models (pre(S_i') \wedge b_k)(\sigma)$. Then

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & \models b_k(\sigma'), \\ & S_i'' \equiv before(S_i', \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

But $<S_i', \sigma'> \rightarrow <S_i^k, \sigma'>$ if $\models b_k(\sigma')$, so

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k', h': \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \qquad <S_1'' \parallel \ldots \parallel S_i^k \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & k' = k+1, h' = h, \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

This implies $\models pre(S_i^k)(\sigma)$, so $(pre(S_i')(\sigma) \wedge b_k) \vdash (pre(S_i^k)(\sigma))$.

(b) Assume that $\models post(S_i^k)(\sigma)$. Then

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv after(S_i^k, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

But

$$\begin{aligned} after(S_i^k, \pi_i) &\equiv after(S_i^k, S_i^k); after(S_i', \pi_i) \\ &\equiv E; after(S_i', \pi_i) \\ &\equiv after(S_i', \pi_i), \end{aligned}$$

so $\models post(S_i')(\sigma)$, and therefore $post(S_i^k)(\sigma) \vdash post(S_i')(\sigma)$.

6. If S_i' is **do** $[]_{k=1:m} b_k \rightarrow S_i^k$ **od**, then we must show that for $k = 1 : m$,

   (a) $(pre(S_i') \wedge b_k)(\sigma) \vdash (pre(S_i^k)(\sigma))$; (b) $post(S_i^k)(\sigma) \vdash pre(S_i')(\sigma)$; and

   (c) $(pre(S_i')(\sigma) \wedge \bigwedge_{k=1}^{m} \neg b_k) \vdash post(S_i')$.

   Therefore

   (a) Assume $\models (pre(S_i') \wedge b_\ell)(\sigma)$. Then

$$\begin{aligned} \models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & \models b_\ell(\sigma'), \\ & S_i'' \equiv before(S_i', \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

   But $before(S_i', \pi_i) \equiv S_i'; after(S_i', \pi_i)$, so

$$<before(S_i', \pi_i), \sigma'> \rightarrow <S_i^\ell; S_i'; after(S_i', \pi_i), \sigma'>$$

   if $\models b_\ell(\sigma')$, and thus since

$$S_i^\ell; S_i'; after(S_i', \pi_i) \equiv before(S_i^\ell, \pi_i),$$

$$\begin{aligned} \models \exists \sigma', \tau, S_1'', \ldots, S_N'', k', h': \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \quad <S_1'' \parallel \ldots \parallel before(S_i^\ell, \pi_i) \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & k' = k + 1, h' = h, \\ & \sigma_i(T_i) = past(\pi_i, h). \end{aligned}$$

   Therefore $\models pre(S_i^\ell)(\sigma)$, so $(pre(S_i') \wedge b_\ell)(\sigma) \vdash (pre(S_i^\ell)(\sigma))$.

(b) Assume that $\models post(S_i^k)(\sigma)$. Then

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & S_i'' \equiv after(S_i^k, \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

But

$$\begin{aligned} after(S_i^k, \pi_i) &\equiv after(S_i^k, S_i'); after(S_i', \pi_i) \\ &\equiv after(S_i^k, S_i^k); S_i'; after(S_i', \pi_i) \\ &\equiv E; S_i'; after(S_i', \pi_i) \\ &\equiv S_i'; after(S_i', \pi_i) \\ &\equiv before(S_i', \pi_i), \end{aligned}$$

so $\models pre(S_i')(\sigma)$, and therefore $post(S_i^k)(\sigma) \vdash pre(S_i')(\sigma)$.

(c) Assume $\models (pre(S_i') \wedge \bigwedge_{k=1}^m \neg b_k)(\sigma)$. Then

$$\begin{aligned}\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \qquad <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & \sigma_i' = \sigma_i, \\ & \models \bigwedge_{k=1}^m \neg b_k(\sigma'), \\ & S_i'' \equiv before(S_i', \pi_i), \\ & \sigma_i(T_i) = past(\pi_i, h).\end{aligned}$$

But $<before(S_i', \pi_i), \sigma'> \rightarrow <after(S_i', \pi_i), \sigma'>$ if $\models \bigwedge_{k=1}^m b_k(\sigma')$, so

$$\models \exists \sigma', \tau, S_1'', \ldots, S_N'', k', h' \quad \models p(\tau),$$
$$<S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^k$$
$$<S_1'' \parallel \ldots \parallel after(S_i', \pi_i) \parallel \ldots \parallel S_N'', \sigma'>,$$
$$\sigma_i' = \sigma_i,$$
$$k' = k + 1, h' = h,$$
$$\sigma_i(T_i) = past(\pi_i, h)].$$

Therefore $\models post(S_i')(\sigma)$, so $(pre(S_i') \wedge \bigwedge_{k=1}^{m} b_k)(\sigma) \vdash (post(S_i')(\sigma))$.

7. If S_i' is $\pi_j!e$, then for all S_j' such that S_j' is a communication command and $S_i' \parallel S_j'$, we must show that

   (a) Only the variables and vector clocks defined in π_i and π_j and the statement labels of S_i', S_j', π_i and π_j are free in $glue(S_i', S_j')$.

   (b) $\neg((pre(S_i') \wedge pre(S_j') \wedge glue(S_i', S_j')) \vdash (T_{i,j} > T_{j,j} \vee T_{j,i} > T_{i,i}))$

   (c) $(pre(S_i') \wedge pre(S_j')) \wedge glue(S_i', S_j')$
   $\vdash (post(S_i') \wedge glue(S_i', S_j'))^{x,T_i,T_j}_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}$

   We do this as follows:

   (a) The functions satisfy condition (a) because all variables used in $glue(S_i', S_j')$ not defined in π_i and π_j are existentially quantified, and the only statement labels used are those of S_i, S_j, π_i and π_j.

(b) Assume that $\models (pre(S_i') \wedge pre(S_j') \wedge glue(S_i', S_j'))(\sigma)$. This implies that

$$\begin{aligned}\exists \sigma', \tau, S_1', \ldots, S_N', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \quad <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & S_i'' \equiv before(S_i', \pi_i) \wedge S_j'' \equiv before(S_j', \pi_j) \\ & \sigma_i' = \sigma_i, \sigma_j' = \sigma_j, \\ & \sigma_i(T_i) = past(\pi_i, h), \\ & \sigma_j(T_j) = past(\pi_j, h).\end{aligned}$$

But if for some h and σ, $S_i' \parallel S_j'$, then $\sigma(T_{i,j}) \not> \sigma(T_{j,j})$ and $\sigma(T_{j,i}) \not> \sigma(T_{i,i})$, so $\neg((pre(S_i') \wedge pre(S_j')) \wedge glue(S_i', S_j')) \vdash (T_{j,i} > T_{j,j} \vee T_{j,i} > T_{i,i}))$.

(c) Assume that $\models (pre(S_i') \wedge pre(S_j') \wedge glue(S_i', S_j'))(\sigma)$. This implies that

$$\begin{aligned}\exists \sigma', \tau, S_1', \ldots, S_N', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_k^h \\ & \quad <S_1'' \parallel \ldots \parallel S_N'', \sigma'>, \\ & S_i'' \equiv before(S_i', \pi_i) \wedge S_j'' \equiv before(S_j', \pi_j) \\ & \sigma_i' = \sigma_i, \sigma_j' = \sigma_j, \\ & \sigma_i(T_i) = past(\pi_i, h), \\ & \sigma_j(T_j) = past(\pi_j, h).\end{aligned}$$

But

$$<S_1'' \parallel \ldots \parallel S_N'', \sigma'> \rightarrow_1^{(e,i,j)} <S_1''' \parallel \ldots \parallel S_N''', \mathcal{M}(S_i' \parallel S_j')(\sigma')>,$$

where S_i''' is $after(s_i, P_i)$, S_j''' is $after(r_j, P_j)$, and for $k \neq i, j$, $S_k''' = S_k''$.

Therefore $<S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_{k+1}^{ho(e,i,j)} <S_1''' \parallel \ldots \parallel S_N''', \mathcal{M}(S_i' \parallel S_j')(\sigma')>$.

Let $\varsigma \equiv \mathcal{M}(S_i' \parallel S_j')(\sigma')$. Then we have

$$\begin{aligned}\exists \varsigma, \tau, S_1'', \ldots, S_N'', k, h: \quad & \models p(\tau), \\ & <S_1 \parallel \ldots \parallel S_N, \tau> \rightarrow_{k+1}^{ho(e,i,j)} \\ & \quad <S_1''' \parallel \ldots \parallel S_N''', \varsigma>, \\ & S_i''' \equiv after(S_i', \pi_i) \wedge S_j''' \equiv after(S_j', \pi_j) \\ & \varsigma \equiv \sigma_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}^{tx,T_i,T_j},\end{aligned}$$

which implies

$$(post(S_i') \wedge glue(S_i', S_j'))(\sigma_{e,synch(i,j,T_i,T_j),synch(i,j,T_i,T_j)}^{x,T_i,T_j}).$$

Thus all 7 parts of the definition of assertion functions are satisfied, and the proof is complete.

■

4.4.3 Relative Completeness of the Proof System

Now that we have shown that *pre*, *post* and *glue* are assertion functions, it is easy to show that our proof system is relatively complete.

Theorem 4.12 (Relative Completeness) *If $\{p\}S\{q\}$ is true in the operational definition of CSP, then if we have a complete deductive system for the natural numbers and any other data types and operations of L, we can prove $\{p\}S\{q\}$ in our proof system.*

- Let $S \equiv S_1 \parallel \ldots \parallel S_N$. By Theorem 4.11, *pre*, *post* and *glue* are assertion functions for $\{p\}S\{q\}$. By Corollary 4.3, this means that we can prove $\{p\}S\{q\}$ in our proof system.

This simple proof completes the long process of showing that what we can prove in our proof system for CSP is true, and that, relative to some complete deductive system,

we can prove anything that is true. Though this exercise has, admittedly, been tedious, we needed to go through it to prove that the restrictions we place on the assertions that we can make in our program annotations impose no loss of power.

It seems reasonable that restricting what we can assert would not affect the soundness of our proofs. It is less obvious that we would not lose any expressive power. With the theorems we have proven in this section, we have answered that doubt. We have shown that, even though we use only causal reasoning, make no global assertions, and avoid the use of auxiliary variables, we need make only recursive assertions to prove that a correct program is, in fact, correct. Our proof system is, therefore, equally as powerful as the standard proof systems for CSP.

Chapter 5

Proof Systems for Asynchronous Programming

5.1 Introduction

We can adapt and extend our work on our CSP proof system to develop proof rules for asynchronous programming. We offer proof systems for two asynchronous message–passing paradigms, unreliable datagrams and virtual circuits. Our motivation remains the same. We want to write proofs of asynchronous programs which help the verifier make a mapping from the code to its specification, while using only assertions which are readily traceable.

In this chapter, we first describe the verification strategy for unreliable *datagrams* suggested by Schlichting and Schneider [SS84]. Their technique uses auxiliary variables and nonlocal predicates. We then define our causal proof system for datagrams. Next, we discuss Schlichting and Schneider's technique for proving programs which use *virtual circuits*, and give our causal axioms and rules for virtual circuits. Finally, we discuss the soundness and relative completeness of our proof systems for datagrams and virtual circuits.

5.2 Untraceable Annotations of Datagram Programs

5.2.1 A Model of Datagram Processing

Low level network protocols may not guarantee that asynchronous messages are delivered in the order sent, or even that all messages sent are delivered. Network software or hardware may discard a corrupt message. Transmission problems may prevent the delivery of a message, while routing algorithms may send messages by different routes so that they arrive out of order.

Some network services use these low level datagrams for interprocess communication. Others, because such a protocol is unsatisfactory for many network clients, use higher level software to transform the unreliable datagrams that the low level software transmits, into reliable, sequenced messages. To be able to show that either sort of network software does what we want it to do, we must be able to prove it correct, so we must be able to write proofs of code which uses unreliable datagrams for interprocess communication.

As Schlichting and Schneider model datagram communication [SS84], a message may be either sent, delivered or received. Execution of a **send** statement sets a message's status to sent. The message may or may not be delivered to its addressee, who can change its status to received by issuing a **receive** command. Delivery is regulated by two rules:

[UD1] A message sent is not always delivered.

[UD2] Messages transmitted from π_i to π_j are not always delivered in the order transmitted.

Process π_i transmits a datagram by executing a **send** statement:

$$\textbf{send } e \textbf{ to } \pi_j$$

which evaluates expression e and then puts its value into the data field of a message addressed to process π_j. The expression e may represent a structured data type. The

sending process does not block. It continues executing while the message is delivered and possibly received. Process π_i may send a message to any process, including itself.

The receiving process π_j inputs the message by executing a **receive** statement:

$$\textbf{receive } x \textbf{ when } \beta$$

The **receive** blocks until the boolean expression β evaluates to true, and then assigns the value of the data field in the incoming message to the local variable x. The boolean expression may reference the contents of the incoming message in addition to local variables. While this would be expensive to implement, it makes the datagram powerful enough to model many communications constructs, such as those which require runtime checking of message types.

To define datagram semantics, Schlichting and Schneider define two auxiliary variables for each process to represent messages addressed to it. The first, σ_{π_i}, is a multiset which holds a copy of each message transmitted to π_i. The other, ρ_{π_i}, is a multiset containing a copy of each message received by π_i. Since a message cannot be received until it has been sent and delivered, we can say

[Unreliable Datagram Network Axiom] $\forall i : \rho_{\pi_i} \subseteq \sigma_{\pi_i}$.

If $\ominus$ denotes the difference operator for multisets, then the multiset $\alpha_{\pi_i} = \sigma_{\pi_i} \ominus \rho_{\pi_i}$ represents messages sent but not received. A "lost" message stays in α_{π_i}. And, since α_{π_i} is unordered, receipts may be out of order.

5.2.2 An Untraceable Proof System for Datagram Processing

Schlichting and Schneider observe that the execution of "**send** e **to** π_i" in their model is operationally equivalent to executing

$$\sigma_{\pi_i} := \sigma_{\pi_i} \oplus e$$

where $\oplus$ is the multiset addition operator. The execution of "**receive** x **when** β" blocks until β is true, then executes the multiple assignment statement

$$x, \rho_{\pi_i} := \text{MTEXT}, \rho_{\pi_i} \oplus \text{MTEXT}$$

where MTEXT is the value of the message data field. The **receive** leaves β true. Since Schlichting and Schneider allow the use of global reasoning, when the **receive** terminates we may make "miraculous" assertions about the state of the sender. The send and receive axioms in their proof system therefore are:

[UD Send Axiom] $\{p^{\sigma_{\pi_i}}_{\sigma_{\pi_i} \oplus e}\}$ **send** e **to** π_i $\{p\}$

[UD Receive Axiom] $\{p\}$ **receive** x **when** β $\{q \wedge \beta\}$

A satisfaction proof is necessary to resolve all miraculous postconditions of **receive** statements, much as in the CSP proof systems of Levin and Gries [LG81] and Schlichting and Schneider. For a **receive** r_i in process π_i to input a message whose data field has value MTEXT, it must be true that:

1. β^m_{MTEXT} holds;
2. MTEXT $\in \alpha_{\pi_i}$.

If the **receive** terminates, the postcondition $(q \wedge \beta)$ holds. This means that a terminating **receive** must begin in one of the set of all states that guarantee that, if the **receive** begins in any of them, it will terminate in a finite time in a state that satisfies $(q \wedge \beta)$. The predicate that represents this set of states, called the *weakest precondition* of the **receive** with respect to its postcondition [Dij76], is

$$\text{wp}(\text{“}x, \rho_{\pi_i} := \text{MTEXT}, \rho_{\pi_i} \oplus \text{MTEXT}\text{”},\ q \wedge \beta) \equiv (q \wedge \beta)^{x, \rho_{\pi_i}}_{\text{MTEXT}, \rho_{\pi_i} \oplus \text{MTEXT}}$$

must be true at the entry to r_i. This means that to justify r_i's postcondition, we must show

[UD Satisfaction Rule] $(pre(r_i) \wedge \beta^m_{\mathrm{MTEXT}} \wedge \mathrm{MTEXT} \in \alpha_{\pi_i}) \vdash q^{x,\rho_{\pi_i}}_{\mathrm{MTEXT},\rho_{\pi_i} \oplus \mathrm{MTEXT}}$

To prove a program correct using these axioms and appropriate axioms and proof rules for the other constructs of the programming language employed in the code, the verifier must show that each process is correct in isolation, that each **receive** statement is satisfied, and that noninterference holds between each assertion and all parallel assignments, sends and receives.

To make establishing satisfaction easier, Schlichting and Schneider suggest that the verifier strengthen the annotation by adding program invariants as conjuncts to each assertion. These invariants either guarantee that one of the conjuncts of the antecedent of the satisfaction rule is false or that the rule's consequent is true. They may falsify $\mathrm{MTEXT} \in \alpha_{\pi_i}$ by ensuring that all messages that could be received have not been transmitted or have been received; falsify $pre(r_i)$ or β^m_{MTEXT} by ensuring that a message sent cannot be received; or ensure that a message can be received and that $q^{x,\rho_{\pi_i}}_{\mathrm{MTEXT},\rho_{\pi_i} \oplus \mathrm{MTEXT}}$ holds.

5.2.3 Our Objections to This Model and Proof System

We can make first the obvious objection that this proof system uses both auxiliary variables and nonlocal assertions, so its annotations are clearly not suitable as directives for execution tracing. Moreover, while global invariants may simplify satisfaction, they force us to use global rather than causal reasoning in every pre– and post–assertion.

We also believe that the global thinking this method uses is an obstacle to the verifier as she tries to make a mapping from program to specification. Because Schlichting and Schneider are wedded to a global view of distributed processing, they must use σ and ρ to represent the network through which processes communicate. They are forced to represent the network because their use of global reasoning requires that they think in terms of concurrency. With asynchronous message passing, it does not help to talk about the sender and receiver concurrently, at the same virtual instant, since the transmission of a message

temporally precedes its receipt. Therefore the model must interpose a representation of the network so that there is *something* whose states both at the time of transmission and at the time of receipt are of interest.

It is not the state of the network, though, in which we are interested. We are interested, as always, in the states of the distributed processes and their causal relationships. Having to make assertions about σ and ρ complicates matters without making our annotations any easier to derive or use. As with CSP, we want to be able to make assertions about processes and pairs of processes, both to make our reasoning during verification less difficult and our trace analysis more tractable.

Having to reason about the state of the network also introduces an unnecessary distinction between proof systems for asynchronous message passing and those for CSP. Synchronous message passing certainly relies on underlying network mechanisms as much as datagram communication does, but we felt no need there to talk about the state of the network. By doing so in this context, we lose any sense of parallel structure between the different communication paradigms. It would be better to use axioms and proof rules which differ only to the extent that the semantics of the paradigms differ, and let the verifier approach programs in similar ways, whether communication is synchronous or asynchronous.

5.2.4 A Sample Annotation Using σ and ρ.

Consider the simple producer/consumer program PRODCON shown in Figure 5.1. The producer, π_1, sends each portion in array A to the consumer, π_2, which stores the portions in its array, B. For each portion, π_1 repeats the **send** until its receipt is acknowledged by π_2. The processes can tell which portion is in a message by checking the sequence number field. The consumer uses the sequence number to tell when to set B[j], and the producer uses it to tell when A[i] has been received, so that it can increment i and transmit the next portion.

```
π1: var A   : array 1..N of portion;
        i   : integer;
        ack : boolean;
        ms1 : record =
              portionnum : integer;
              x : portion
              end;
    i := 1;
    while i < N + 1 do
        ack := false;
        while not(ack) do
              ms1.portionnum := i;
              ms1.x := A[i];
              S1:: send ms1 to π2;
              if Ready → R1:: receive ms1 when true;
              [] Timeout → ms1.portionnum := 0
              fi;
              if ms1.portionnum = i then ack := true
        od
        i := i + 1
    od

π2: var B   : array 1..N of portion;
        j   : integer;
        ok  : boolean;
        ms2 : record =
              portionnum : integer;
              x : portion
              end;
    j := 1;
    while j < N + 1 do
        ok := false;
        while not(ok) do
              R2:: receive ms2 when true;
              S2:: send ms2 to π1;
              if ms2.portionnum = j then
                B[j] := ms2.x;
                ok := true
              fi
        od
        j := j + 1
    od
```

Figure 5.1: Producer/consumer program PRODCON, with datagrams.

Since messages can be lost, we need to be able to stop a receiving process from blocking if no message is delivered in some reasonable period, where "reasonable" is defined by the programmer at compile time. We therefore define a function *Timeout* which blocks until that reasonable period has expired, and then returns true. We also define function *Ready*, which blocks until a message is delivered to the receiver, and then returns true. We can use Timeout and Ready as guards in an alternation statement to accept an incoming message without blocking indefinitely. In this example, the producer uses the Timeout function to avoid blocking forever in case messages are lost. Obviously, we have given only informal semantics for Timeout and Ready, which depend on low level, system–specific functions. We do not need to be more precise, because we really only care that after the **if** statement in π_i, π_i has some appropriate value for variable ms1.portionnum.

In Figures 5.2 and 5.3, we show an annotation of this program in the style of Schlichting and Schneider. Demonstrating noninterference is straightforward, and showing satisfaction is trivial for both **receive** R_1 in π_1 and **receive** R_2 in π_2, since the global invariant I describes how the elements of array A are sent and received. Tracing an execution of PRODCON using this annotation, on the other hand, would be very difficult since every assertion references I. Moreover, the use of σ and ρ does little to clarify the action of the processes. We care mostly about assertion P in the annotation of π_2, and talking so much about the history of what is sent to and received by π_1 and π_2 is confusing rather than edifying.

PRODCON is not an example contrived to show deficiencies in Schlichting and Schneider's proof system. It is typical of programs which use datagrams. Other datagram programs may use a sliding window protocol or some other algorithm to allow more asynchrony between sender and receiver, but essentially they are similar, since datagrams are rarely used for anything but low-level data transmission. The problem is that send and receive multisets cannot tell us much about whether a datagram program is doing its job. Attention to the underlying network unnecessarily complicates the semantics of datagram

```
π1: var A   : array 1..N of portion;
        i   : integer;
        ack : boolean;
        ms1 : record =
              portionnum : integer;
              value : portion
              end;
    i := 1;
    { i = 1 ∧ I ∧ σπ2 = Φ }
    while i < N + 1 do
        { I }
        ack := false;
        { I ∧ ack = false }
        while not(ack) do
            { I ∧ ack = false }
            ms1.portionnum := i;
            { I ∧ ack = false ∧ ms1.portionnum = i }
            ms1.value := A[i];
            { I ∧ ack = false ∧ ms1.portionnum = i ∧ ms1.value = A[i] }
            S1: send ms1 to π2;
            { I ∧ (i,A[i]) ∈ σπ2 ∧ ack = false ∧ ms1.portionnum = i ∧ ms1.value = A[i] }
            if Ready → R1:: receive ms1 when true;
            [] Timeout → ms1.portionnum := 0
            fi
            { I ∧ (i,A[i]) ∈ σπ2 ∧ ack = false }
            if ms1.portionnum = i then ack := true
            { I ∧ (i,A[i]) ∈ σπ2 }
        od
        { I ∧ (i,A[i]) ∈ σπ2 ∧ (i,A[i]) ∈ ρπ1 }
        i := i + 1
        { I }
    od
    { i = N+1 ∧ I }
```

$$I \equiv \forall k\colon 1 \le k < i : ((k,A[k]) \in \sigma_{\pi_2} \wedge (k,A[k]) \in \rho_{\pi_2} \wedge (k,A[k]) \in \sigma_{\pi_1} \wedge (k,A[k]) \in \rho_{\pi_1}) \\ \wedge \forall\, m : m \in \sigma_{\pi_2} : (m.value = A[m.portionnum])$$

Figure 5.2: Annotation of producer using σ and ρ .

```
π2: var B   : array 1..N of portion;
      j     : integer;
      ok    : boolean;
      ms2   : record
                portionnum : integer;
                value : portion
                end;
    j := 1;
    { j = 1 ∧ I ∧ ρπ2 = Φ }
    while j < N + 1 do
        { I ∧ P}
        ok := false;
        { I ∧ P ∧ ok = false }
        while not(ok) do
            { I ∧ P ∧ ok = false }
            R2: receive ms2 when true;
            { I ∧ P ∧ ok = false ∧ ms2 ∈ ρπ2}
            S2: send ms2 to π1;
            { I ∧ P ∧ ok = false }
            if ms2.portionnum = j then
              B[j] := ms2.value;
              ok := true
              { I ∧ (j,A[j]) ∈ ρπ2 ∧ (j,A[j]) ∈ σπ1 ∧ P ∧ B[j]=A[j] ∧ ok = true }
            fi
        od
        { I ∧ (j,A[j]) ∈ ρπ2 ∧ (j,A[j]) ∈ σπ1 ∧ P ∧ B[j]=A[j] ∧ ok = true }
        j := j + 1
        { I ∧ P}
    od
    { j = N+1 ∧ I ∧ P}
```

$$P \equiv (\forall k\colon 1 \leq k < j\colon A[k]=B[k])$$

Figure 5.3: Annotation of consumer using σ and ρ .

communication, without making it easier to prove programs correct.

5.3 Causal Annotations of Programs Which Use Datagrams

5.3.1 Causal Semantics and Proof Rules for Datagrams

Datagram semantics would be clearer and our proofs simpler if we could define **send** and **receive** in terms of the causally related states of the sender and the receiver, as we did with CSP, without recourse to some occult model of the communication medium. If we exploit causal reasoning in a model of datagram processing, we can define a semantics for datagram sends and receives using neither auxiliary variables nor global reasoning.

Operationally, we can say that a datagram **send** in π_i increments the ith element in π_i's vector clock T_i by some $d > 0$, and then transmits a message addressed to π_j, timestamped with the value of T_i, whose data field has value e. Except for the clock update, the **send** has no effect on the state of π_i, and does not block the progress of its execution.

The clock update for an asynchronous **send** affects only the ith element of T_i because causality flows only from the sending process to the receiving process, and not both ways as it does in a synchronous communication. Sender and receiver do not synchronize at an asynchronous communication as they do at a synchronous one. The **send** has causal precedence over the commands that the receiver executes after its **receive** in both asynchronous and synchronous communication, but the **receive** has causal precedence over the statements that the sender executes after the **send** only if the communication is synchronous.

A **receive** command in π_i blocks until β is true. It can terminate only when it communicates with a **send** to π_i, never in isolation. With β true, a receipt of a message

with data field MTEXT and timestamp $\vec{X}$ executes the multiple assignment

$$x, T_i := \text{MTEXT}, asynch_r(i, T_i, \vec{X})$$

where $asynch_r$, the update function for asynchronous receives, is

$$asynch_r(i, \vec{Y}, \vec{X}) \equiv \sup(<y_1, \ldots, y_i + d, \ldots, y_n>, \vec{X}),$$

for some $d > 0$. This update establishes the causal precedence of the **send** over the **receive**, since it will be the case that if the **receive** terminates, $T_i > \vec{X}$. Since datagrams may be delivered out of order, it may be the case that the only change to T_i is the addition to its ith element. A previous receipt of a message sent after the message currently received will have made each element of T_i greater than the corresponding element in $\vec{X}$.

With these definitions of command semantics, we can give our send and receive axioms:

[UD Send Axiom] $\{p^{T_{ii}}_{T_{ii}+d}\}$ **send** e **to** π_j $\{p\}$.

[UD Receive Axiom] $\{p\}$ **receive** x **when** β $\{true\}$.

This UD Receive Axiom is very similar to the CSP receive axiom, since neither **receive** can terminate in isolation, but the send axioms are different. Since a datagram **send** is nonblocking, we do not have to use a satisfaction rule for it as we do for a CSP **send**.

To draw a more useful postassertion than *true* for a datagram **receive**, though, we must use a satisfaction rule that pairs it with all sends which could have transmitted its message. For this purpose, we use our **Datagram Satisfaction Rule**:

$$\frac{\begin{array}{l}\{p\}R_i :: \textbf{receive } x \text{ when } \beta\{true\}, p \vdash T_i = \vec{X}, \\ \forall S_j : S_j :: \textbf{send } e \text{ to } \pi_j \wedge \{q\}S_j\{r\} \wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{Y_i} > \vec{X_i})) : \\ \qquad (p \wedge q \wedge glue(S_j, R_i)) \vdash (s \wedge glue(s_j, r_i))^{x, T_{jj}, T_i}_{e, T_{jj}+d, asynch_r(i, T_i, T_j)}\end{array}}{\{p\}R_i :: \textbf{receive } x \text{ when } \beta\{s \wedge \beta\}}$$

where only the variables and vector clocks of π_i and π_j and the statement labels of R_i and S_j are free in $glue(S_j, R_i)$.

This is much like our satisfaction rule for CSP, as we would expect since the semantics of the two receives are similar. To derive the postassertion of a **receive**, we must consider the preassertions of all potentially communicating sends and the glue assertions which link them to the **receive**, as well as the **receive**'s preassertion. We do not need to consider the state of the network; we can find what we need just by looking for the possible causal relationships between the receiver and the potential senders.

The difficulty is that since datagrams can be lost or delivered out of order, and since sender and receiver do not synchronize, we cannot rule out as many sends as we can in CSP. In CSP, we are only interested in those pairs of sends and receives which could execute in parallel. Here, we can only exclude a **send** if we can show that the sender has knowledge of this or a later message receipt by the receiver. If this is the case, the **send** necessarily follows the **receive** in the program's causal order, and the pair could not communicate.

In general, the postassertions we can derive for datagram receives are weak compared to those we are used to from CSP proofs. In our example program PRODCON, for instance, a **receive** in process π_2 when j is 5 may be of a message with any portion number from 1 through 5. Our assertions must reflect this, but they should do so without going into extraneous detail.

5.3.2 A Causal Annotation of PRODCON

In Figures 5.4 and 5.5, we give a causal annotation of PRODCON.

W, X, Y and $\vec{Z}$, used in the glue assertions and the annotation of π_2, are existentially quantified over the entire program annotation. Y represents the unknown, but fixed, value of the portion number in an incoming message in π_2. $\vec{Z}$ is a vector of N portion values. When π_2 receives a message, we say that its value is the Yth element of $\vec{Z}$. W and X represent the values of vector clock elements.

Showing satisfaction for R_2 involves showing that

```
π1: var A   : array 1..N of portion;
        i   : integer;
        ack : boolean;
        ms1 : record =
              portionnum : integer;
              value : portion
              end;
    i := 1;
    { i = 1 }
    while i < N + 1 do
        ack := false;
        { ack = false }
        while not(ack) do
            { ack = false }
            ms1.portionnum := i;
            { ack = false ∧ ms1.portionnum = i }
            ms1.value := A[i];
            { ack = false ∧ ms1.portionnum = i ∧ ms1.value = A[i] }
            S1: send ms1 to π2;
            { ack = false ∧ ms1.portionnum = i ∧ ms1.value = A[i] }
            if Ready → R1:: receive ms1 when true;
            [] Timeout → ms1.portionnum := 0
            fi
            { ack = false }
            if ms1.portionnum = i then ack := true
        od
        i := i + 1
    od
    { i = N+1 }
```

Figure 5.4: Causal annotation of the producer.

```
π2: var B   : array 1..N of portion;
       j    : integer;
       ok   : boolean;
       ms2  : record
                portionnum : integer;
                value : portion
                end;
    j := 1;
    { j = 1 ∧ P }
    while j < N + 1 do
        ok := false;
        { P ∧ ok = false }
        while not(ok) do
            { P ∧ ok = false }
            R2: receive ms2 when true;
            { P ∧ ok = false ∧ ms2.portionnum = Y ∧ ms2.value = Z_Y }
            S2: send ms2 to π1;
            { P ∧ ok = false ∧ ms2.portionnum = Y ∧ ms2.value = Z_Y }
            if ms2.portionnum = j then
              B[j] := ms2.value;
              ok := true
              { P ∧ ok = true ∧ j = Y ∧ B[j] = Z_Y }
            fi
        od
        { P ∧ ok = true ∧ j = Y ∧ B[j] = Z_Y }
        j := j + 1
        { P ∧ ok = true }
    od  { j = N+1 ∧ P }
```

$$P \equiv (\forall k\colon\ 1 \le k < j\colon\ B[k]{=}\vec{Z}_k)$$

$$glue(S_1, R_2) \equiv (after(R_2) \wedge T_{2,1} = W \wedge \text{ ms2.portionnum} = Y)$$

$$\vdash [(after(S_1) \wedge T_{1,1} = W) \vdash (i = Y \wedge \forall k : 1 \le k \le i : A[k] = \vec{Z}_k]$$

$$glue(S_2, R_1) \equiv (after(R_1) \wedge T_{1,2} = X \wedge \text{ ms2.portionnum} = Y)$$

$$\vdash [(after(S_2) \wedge T_{2,2} = X) \vdash \text{ ms1.portionnum} = Y]$$

Figure 5.5: Causal annotation of consumer.

$$(\text{ack} = \text{false} \wedge \text{ms1.portionnum} = \text{i} \wedge \text{ms1.value} = \text{A[i]} \wedge \text{P} \wedge \text{ok} = \text{false} \wedge \mathit{glue}(S_1, R_2))$$
$$\vdash (\text{P} \wedge \text{ok=false} \wedge \text{ms2.portionnum=}Y \wedge \text{ms2.value=}\vec{Z}_Y \wedge \mathit{glue}(S_1, R_2))^{\text{ms2},T_{1,1},T_2}_{\text{ms1},T_{1,1}+1,T_2'}$$

where $T_2' = \mathit{asynch}_r(2, T_2, T_1)$, and that

$$(\text{ack} = \text{false} \wedge \text{ms1.portionnum} = \text{i} \wedge \text{ms1.value} = \text{A[i]} \wedge \text{P}$$
$$\wedge\ \text{ok} = \text{false} \wedge \text{ms2.portionnum} = Y \wedge \text{ms2.value} = \vec{Z}_Y \wedge \mathit{glue}(S_2, R_1))$$
$$\vdash (\ \text{ack} = \text{false} \wedge \mathit{glue}(S_1, R_2))^{\text{ms1},T_{2,2},T_1}_{\text{ms2},T_{2,2}+1,T_1'}$$

where $T_1' = \mathit{asynch}_r(1, T_1, T_2)$. Both implications are straightforward.

The assertion $\mathit{glue}(S_2, R_1)$ states that when π_1 receives an acknowledgement message, the portion number in the message is in fact that of an acknowledgement previously transmitted by π_2. The assertion $\mathit{glue}(S_1, R_2)$ tells us that if π_2 receives a message with portion number Y, then at the exit point of the transmission of that message in π_1, the value of variable i was Y, and that for indices 1 through i, the elements of A had the same values as the corresponding elements of $\vec{Z}$. But assertion P in π_2 tells us that for indices 1 through j, the elements of B also have the same values as the corresponding elements of $\vec{Z}$. As π_2 iterates through its outer loop, it assigns more values to B. If the loop terminates, it will set all N elements of B equal to the corresponding elements of $\vec{Z}$. Since it must have received a message with portion number N to terminate, we can use $\mathit{glue}(S_1, R_2)$ to deduce that the elements of A and B are equal when π_2 terminates.

Without talking about the state of the network, and comparing the states of the two processes only in the glue assertions, our annotation highlights the causal relationships between the states of the producer and consumer that guarantee that the values in array A are transmitted and received correctly. We have no need to establish any concurrent, global relationships, nor any need to make an assertion which is difficult to record in a trace log or to analyze after the program terminates. Since our process annotations reference

only local variables, we do not need to write a noninterference proof. We could take the same approach with any program which used datagrams, and so avoid the problems inherent in Schlichting and Schneider's model.

5.4 Untraceable Annotations of Virtual Circuit Programs

5.4.1 A Model of Virtual Circuit Processing

We can apply a similar analysis to Schlichting and Schneider's model and proof system for virtual circuits. A virtual circuit is a unidirectional communications construct, connecting two processes, in which two properties hold:

[VC1] A message sent is delivered.

[VC2] Messages are delivered in the order sent.

The **send** and **receive** statements for virtual circuits are "**send** e **on** V" and "**receive** x **on** V", where V is the name of a communications channel which satisfies VC1 and VC2, e is an expression, and x is a structured or elementary variable defined in the receiving process.

As with unreliable datagrams, Schlichting and Schneider use auxiliary variables to define the semantics of **send** and **receive** commands for virtual circuits. Here, σ_V and ρ_V record the *sequence* of messages that have been sent and received, respectively, on V. By VC2, we see that the axiom

[VC Network Axiom] $\forall V : V$ is a virtual circuit $: \rho_V \leq \sigma_V$

must be true, where $s \leq t$ indicates that s is a prefix of t. Thus a **send** on V evaluates expression e and appends its value to the end of σ_V. A **receive** on V assigns the value of the first previously unreceived element of σ_V to x, and appends a copy of the element to the end of ρ_V.

The send and receive axioms for Schlichting and Schneider's proof system are

[VC Send Axiom] $\{p^{\sigma_V}_{\sigma_V \circ e}\}$ send e **on** V $\{p\}$,

[VC Receive Axiom] $\{p\}$ **receive** x **on** V $\{q\}$.

where $s \circ x$ is the sequence obtained by adding element x to the end of sequence s.

As before, nonlocal variables may appear in the annotation of a process, and the post-assertion of the **receive** statement is miraculous. To use these axioms in a valid proof, therefore, we must show that no **send**, **receive** or assignment interferes with any parallel assertion, and we must show satisfaction for each **receive** command. If $s - t$ denotes the sequence we get by removing prefix t from sequence s, and $head(s)$ is the first element in s, then we satisfy postassertion q of **receive** r_i with preassertion p by using the VC **Satisfaction Rule:**

$$\mathrm{Sat}_{\mathrm{asynch}}(r) : (p \wedge (\sigma_V - \rho_V) \neq \Phi \wedge \mathrm{MTEXT} = head(\sigma_V - \rho_V)) \vdash q^{r,\rho_V}_{\mathrm{MTEXT},\rho_V \circ \mathrm{MTEXT}}.$$

Schneider and Schlichting's send and receive axioms and satisfaction rule for virtual circuits are very similar to those in their proof system for datagrams. They differ mainly in that with virtual circuits σ and ρ are sequences, while with datagrams they were unordered multisets. With datagrams, MTEXT could be any message sent but not received, but here, it can only be the earliest message with that status.

5.4.2 An Annotation Using σ and ρ.

To illustrate the use of this system, we consider a virtual circuit implementation of the producer/consumer paradigm. Program VCPRODCON is shown in Figure 5.6. Again, the producer, π_1, sends each portion in array A to the consumer, π_2. Since it uses virtual circuit V, π_1 need never repeat a **send** or await an acknowledgement. Similarly, the consumer can rely on ordered message delivery. It simply stores the incoming data in array B. This makes the code of both processes simpler than in the datagram implementation of the paradigm.

```
π1: var A : array 1..N of portion;
        i : integer;
    i := 1;
    while i < N + 1 do
        send A[i] on V;
        i := i + 1
    od

π2: var B : array 1..N of portion;
        j : integer;
    j := 1;
    while j < N + 1 do
        r: receive B[j] on V;
        j := j + 1
    od
```

Figure 5.6: Program VCPRODCON, using virtual circuits.

In Figure 5.7 we give an annotation of VCPRODCON using σ_V and ρ_V. Non-local references in assertions in the consumer's annotation would make tracing difficult, and the use of σ_V and ρ_V would force us to change the code before testing. And, as with the Schlichting/Schneider annotation of the datagram implementation, we are led to reason about the state of the network rather than about the causal relations between the processes.

5.5 Causal Annotations of Virtual Circuit Programs

5.5.1 Causal Semantics and Proof Rules for Virtual Circuits

As with datagrams, we believe that defining the semantics of the **send** and **receive** statements for virtual circuits in terms of the state of the underlying network is unnecessary if we use causal, rather than global, reasoning in our model of processing.

We can say that, operationally, a virtual circuit **send** in π_i increments the ith element in π_i's vector clock T_i by some $d > 0$, and transmits a message, timestamped with the

π_1: var A : **array** 1..N **of** portion;
i : **integer**;
i := 1;
{ i = 1 $\wedge$ $\sigma_V = \Phi$ }
while i < N + 1 **do**
{ I $\wedge$ i = $|\sigma_V| + 1$ }
send A[i] **on** V;
{ I $\wedge$ i+1 = $|\sigma_V| + 1$ }
i := i + 1
od
I: ($\forall$k: 1 $\leq$ k $\leq$ $|\sigma_V|$: $\sigma_V[k] = A[k]$)

π_2: **var** B : **array** 1..N **of** portion;
j : **integer**;
j := 1;
{ I $\wedge$ j = 1 $\wedge$ $\rho_V = \Phi$ }
while j < N + 1 **do**
{ I $\wedge$ ($\forall$k: 1 $\leq$ k < j: A[k]=B[k] $\wedge$ j = $|\rho_V| + 1$ }
r: **receive** B[j] **on** V;
{ I $\wedge$ ($\forall$k: 1 $\leq$ k $\leq$ j: A[k]=B[k] $\wedge$ j = $|\rho_V|$ }
j := j + 1
od

$\mathrm{Sat}_{\mathrm{asynch}}(r)$:
(I $\wedge$($\forall$k: 1 $\leq$ k < j: A[k]=B[k]) $\wedge$ j = $|\rho_V| + 1 \wedge \sigma_V - \rho_v \neq \Phi \wedge$ MTEXT=hd($\sigma_V - \rho_V$))
$\supset$ (I $\wedge$ ($\forall$k: 1 $\leq$ k $\leq$ j: A[k]=B[k] $\wedge$ j = $|\rho_V|$)$^{B[j],\rho_V}_{MTEXT,\rho_V \cup <MTEXT>}$

Figure 5.7: Annotation of VCPRODCON using σ_V and ρ_V .

value of T_i, whose data field has value e. Except for the clock update, a **send** has no effect on the state of π_i, and does not block the progress of its execution.

A receipt by π_i of a message with data field MTEXT and timestamp $\vec{X}$ executes the multiple assignment

$$x, T_i := \text{MTEXT}, \mathit{asynch}_r(i, T_i, \vec{X})$$

A **receive** command blocks until a message is available for receipt on virtual circuit V. It can terminate only when paired with a **send** on V, never in isolation.

With these definitions of command semantics, we can give our send and receive axioms:

[VC Send Axiom] $\{p^{T_{ii}}_{T_{ii}+d}\}$ **send** e **on** V $\{p\}$.

[VC Receive Axiom] $\{p\}$ **receive** x **on** V $\{\mathit{true}\}$.

These are similar to the corresponding axioms for datagrams. Here, as with datagrams, we do not need to use satisfaction to derive the postcondition of the nonblocking **send**. The distinction between the two communication paradigms is revealed in the satisfaction rule that we must use to draw a more useful postassertion for a **receive**. The **VC Satisfaction Rule** is:

$$\frac{\begin{array}{l}\{p\}R_i :: \textbf{ receive } x \textit{ on } V \ \{\mathit{true}\}, p \vdash T_i = \vec{X}, \\ \forall S_j : S_j :: \textbf{send } e \text{ to } \pi_j \wedge \{q\}S_j\{r\} \wedge q \vdash T_j = \vec{Y} \wedge \neg(\vdash (\vec{Y_i} > \vec{X_i} \vee \vec{X_j} > \vec{Y_j})) : \\ \qquad (p \wedge q \wedge \mathit{glue}(S_j, R_i)) \vdash (s \wedge \mathit{glue}(s_j, r_i))^{x, T_{jj}, T_i}_{e, T_{jj}+d, \mathit{asynch}_r(i, T_i, T_j)}\end{array}}{\{p\}R_i :: \textbf{receive } x \text{ on } V \ \{s \wedge \beta\}}$$

where only the variables and vector clocks of π_i and π_j and the statement labels of R_i and S_J are free in $\mathit{glue}(S_j, R_i)$. With datagrams, we could only rule out those send statements that we could show necessarily followed the **receive** in the program's causal order. Here, we can exclude a **send** if the preconditions of the **send** and **receive** show either that the receiver has knowledge of a later message transmission by the sender or that the sender has knowledge of a later message receipt by the receiver. With virtual circuits, we can rely

on ordered and guaranteed message delivery, and we can use this to rule out more sends and so make our postassertions stronger than we could with datagrams. This satisfaction rule reflects the fact that virtual circuits are an intermediate case between the very loosely structured world of the unreliable datagram and the tightly synchronized world of CSP.

5.5.2 A Causal Annotation of VCPRODCON

In Figure 5.8 we give a causal annotation of VCPRODCON. $\vec{X}$, $\vec{Y}$ and Z are existentially quantified over the entire program annotation. $\vec{X}$ and $\vec{Y}$ are vectors of N portion values. Z represents the value of an vector clock element.

The implication required for satisfaction is straightforward. Assertion *glue*(s,r) states that if after a **receive** in π_2, T_2 has value $[X, X]$ and the first X elements of B have the same value as the first X elements of $\vec{Y}$, then after the communicating **send**, that which left $T_1 = [X, 0]$, the first X elements of A also have the same value as the first X elements of $\vec{Y}$. As the loop iterates in π_2, more elements are assigned to B, and we can deduce that when the loop terminates all N elements of A have been transferred to B.

Again, as we did with the datagram implementation, we have written a suitable annotation without reference to the network or to auxiliary variables, and without global assertions. No noninterference proof is required. The annotation is traceable, and is clearer than that which uses σ_V and ρ_V. With virtual circuits as with datagrams, talking about the network adds a layer of complexity, without adding any power.

5.6 Soundness and Completeness of These Proof Systems

To show that our proof systems for datagrams and virtual circuits is sound and relatively complete, we could start from first principles as we did in Chapter 4. As we saw in Chapter 4, that is a very tedious process, so instead, we will show how both datagrams and virtual circuits can be implemented using CSP sends and receives. We can then see

π_1: **var** A : **array** 1..N **of** portion;
i : **integer**;
i := 1;
{ $I \wedge$ i = 1 }
while i < N + 1 **do**
{ $I \wedge$ i < N+1 }
s: **send** A[i] **on** V;
{ $I \wedge$ i < N+1 }
i := i + 1
od
{ $I \wedge$ i = N + 1}
$I \equiv (\forall k : 1 \leq k \leq$N: A[k]=$X_k$)

π_2: **var** B : **array** 1..N **of** portion;
j : **integer**;
j := 1;
{ j = 1 }
while j < N + 1 **do**
{ j < N+1 $\wedge \forall k : 1 \leq k < j$: B[k]=$Y_k$)}
r: **receive** B[j] **on** V;
{ $\forall k : 1 \leq k \leq j$: B[k]=$Y_k$)}
j := j + 1
od
{ j = N + 1 $\wedge (\forall k : 1 \leq k \leq$ N: B[k]=Y_k)}

$glue$(s,r) $\equiv$ ($after$(r)$\wedge$ $T_2 =< Z, Z > \wedge (\forall k : 1 \leq k \leq Z$: B[k]=$Y_k$)) $\vdash$
(($after$(s) $\wedge$ I $\wedge$ $T_1 =< Z, 0 >$) $\vdash$ ($\forall$k: 1$\leq$k$\leq$ Z: A[k]=Y_k))

Satisfaction $\equiv$ (i < N+1 $\wedge$ I $\wedge$ j < N+1 $\wedge \forall k : 1 \leq k < j$: B[k]=$Y_k$) $\wedge$ $glue$(s,r)) $\vdash$
($\forall k : 1 \leq k \leq j$: B[k]=$Y_k$ $\wedge$ I $\wedge$ $glue$(s,r))$^{B[j],T_{1,1},T_2}_{A[i],T_{1,1}+1,asynch_r(2,T_2,T_1)}$

Figure 5.8: Causal annotation of VCPRODCON.

that the causal asynchronous send and receive axioms and satisfaction rules are faithful translations of their CSP counterparts.

Suppose we have a parallel program P with processes $\pi_1 \ldots \pi_n$. We can implement datagram sends and receives in P as CSP sends and receives to and from a buffer process π_k, which is not a constituent process of P. For a send "S_i:: **send** e **to** π_j" in process π_i and **receive** "R_j:: **receive** x **when** β" in process π_j, we execute the CSP code annotated in Figure 5.9.

VC_i and VC_j are the timestamp vectors that will be returned to and accessible by π_i and π_j; T_i, T_j and T_k are used only by the underlying system which implements P. The operator $\oplus$ is the multiset addition operator. Function *empty(s)* returns true if multiset $s \neq \Phi$, while *extractmem(s)* selects nondeterministically, returns and deletes an element from s.

On a **send** in π_i, we update the ith element in the local clock vector and send the message and clock value to the buffer process. The buffer may nondeterministically choose one of four actions: it may accept an incoming message and insert the data and clock values into a multiset buffer which stores data/clock tuples; it may extract a tuple from the buffer, if the buffer is not empty; it may transmit a data/clock tuple to π_j; or it may execute a skip. Since the buffer is a multiset, messages may be delivered out of order. Messages may be "lost" if the buffer process extracts more than one tuple from the storage buffer before transmitting to the receiver. The receiver, π_j, accepts input from π_k. If the incoming message and π_j's local state satisfy β, π_j assigns the message's data value to its variable x and updates clock VC_j to reflect the indirect, and therefore asynchronous, receipt from π_i. Otherwise, the message is discarded and "lost."

Since the postassertions we need for the communication commands are so weak, the use of the CSP satisfaction rule is easy, and we will leave satisfaction to the reader. The three glue assertions show how data and clock values move from π_i to π_j. The first two link directly communicating statements, and we use them and the process annotations to

π_k:: store := ϕ;
  do [] r_k:: π_i?(indata,intime) → { indata = Y ∧ intime = $\vec{Z}$ ∧ store = M }
                insert(store,(indata,intime))
                { indata = Y ∧ intime = $\vec{Z}$ ∧ store = $M \oplus (Y, \vec{Z})$ }
    [] s_k:: not((outdata,outtime) = NULL);
        { outdata = A ∧ outtime = $\vec{B}$ }
        π_j!(outdata,outtime) → (outdata,outtime) := NULL
        { (outdata,outtime) = NULL }
    [] not(empty(store)) → (outdata,outtime) := extractmem(store)
      { (outdata,outtime) ≠ NULL }
    [] true → skip
  od;

S_i:: { $VC_i = \vec{C} = [c_1, \ldots, c_n]$ ∧ $e = W$ }
  $VC_{i,i} := VC_{i,i} + d$;
  { $VC_i = [c_1, \ldots, c_i + d, \ldots, c_n]$ ∧ $e = W$ }
  s_i:: π_k!(e, VC_i);
  { $VC_i = [c_1, \ldots, c_i + d, \ldots, c_n]$ ∧ $e = W$ }

R_j:: { $VC_j = \vec{L}$ }
  ok := false;
  while not(ok) **do**
      r_j:: π_k?(tempdata,temptime);
      { tempdata = D ∧ temptime = $\vec{E}$ }
      if β
        x := tempdata;
        $VC_{j,j} := VC_{j,j} + d$;
        VC_j := sup (VC_j, temptime);
        ok := true
      fi
  od
  { x = D ∧ $VC_j = asynch_r(j, \vec{L}, \vec{E})$ }

$glue(s_i, r_k) \equiv (after(r_k) \wedge T_k = \vec{F} \wedge \text{indata} = Y \wedge \text{intime} = \vec{Z})$
  $\vdash ((after(s_i) \wedge T_i = \vec{F}) \vdash (\text{e} = Y \wedge VC_i = \vec{Z}))$
$glue(s_k, r_j) \equiv (after(r_j) \wedge T_j = \vec{G} \wedge \text{tempdata} = D \wedge \text{temptime} = \vec{E})$
  $\vdash ((after(s_k) \wedge T_k = \vec{G}) \vdash (\text{outdata} = D \wedge \text{outtime} = \vec{E}))$
$glue(s_i, r_j) \equiv (after(r_j) \wedge \text{tempdata} = D \wedge \text{temptime} = \vec{E})$
  $\vdash ((after(s_i) \wedge VC_i = \vec{E}) \vdash (\text{e} = D))$

Figure 5.9: Annotation of an implementation of datagram communication using CSP.

derive the third, which shows that π_i and π_j do communicate, albeit indirectly.

The glue assertions and the postassertions of send s_i and of the while loop in π_j show that data values are transmitted and clocks updated as required by our datagram axioms and satisfaction rule. The exclusion in the datagram satisfaction rule of **send** statements which necessarily follow the **receive** in the program's causal order is justified by the implementation's use of the multiset storage buffer. Any element in the buffer can be selected for transmission to the receiver, but the receiver cannot be sent a message which has not yet been inserted into the buffer.

We can derive the causal datagram send axiom from the precondition of the implementation of S_i, an application of the assignment rule to account for the increment of $VC_{i,i}$, and the CSP satisfaction proof for s_i. We do not need to show satisfaction for datagram sends since we can derive the satisfaction proof for the CSP implementation from the send axiom.

Similarly, we can derive our datagram receive axiom from the precondition of the implementation of R_j, the satisfaction proof for r_j and applications of the assignment and alternation rules. This process yields the postcondition of the implementation of R_j, which implies *true*, which is what we need for the postcondition required by our axiom. To derive the datagram satisfaction rule, we must follow a chain of reasoning through the code of S_i, π_k and R_j. We note that from satisfying r_k and from the postcondition of the insert statement which follows r_k, we know that the data and clock values sent by S_i are stored in multiset store. Then, from the semantics of the extractmem function and the satisfaction of r_j, we can deduce that data and clock values previously sent by s_i have been received in R_j. Appropriate use of the assignment and alternation rules, as before, yields the postcondition of R_j. We can then use $glue(s_i, r_j)$ to tie the postcondition assertion to the state of the sender as required by the satisfaction rule.

We implement virtual circuit sends and receives in a similar way. In this case, the buffer process must maintain an ordered sequence of messages. When it accepts an incoming

message, it must append its value to the end of the sequence. When it transmits, it must send the value at the front of the sequence and then behead the sequence. The annotated code for implementations of a **send** S_i in process π_i on virtual circuit V, a **receive** R_j on V in π_j, and the buffer process π_k, is shown in Figure 5.10. As before, π_k is not a process of the program P of which π_i and π_j are constituents, but of the underlying implementation.

The operator $\circ$ is the sequence concatenation operator. Function *empty(s)* returns true if sequence $s \neq \Phi$. Function *head(s)* returns the first element of s, and *behead(s)* removes the first element of s. Again, the use of satisfaction is relatively simple, and we use the same glue assertions in the same way as in the annotation of the datagram implementation. In the virtual circuit satisfaction rule, we excluded sends if either the sender has knowledge of the **receive** in question, or if the receiver has knowledge of the **send**. This policy is justified by the implementation's use of a sequenced message buffer, which guarantees that messages will be delivered in order.

As with datagrams, it is straightforward to derive the virtual circuit send and receive axioms and satisfaction rule from the annotations of their implementations. Thus we can use CSP to implement both datagrams and virtual circuits in a way that shows that our proof rules and axioms are valid translations of their CSP counterparts. We can be be confident, therefore, from the soundness and relative completeness of the CSP proof system, that our proof systems for the asynchronous communication paradigms are also sound and relatively complete.

π_k:: store := ϕ;
do [] r_k:: π_i?(indata,intime) → { indata = Y ∧ intime = $\vec{Z}$ ∧ store = M }
append(store,(indata,intime))
{ indata = Y ∧ intime = $\vec{Z}$ ∧ store = $M \circ (Y, \vec{Z})$ }
[] s_k:: not(empty(store));
{ store = $(A, \vec{B}) \circ H$ }
π_j!head(store) → behead(store)
{ store = H }
[] true → skip
od;

S_i:: { $VC_i = \vec{C} = [c_1, \ldots, c_n] \wedge e = W$ }
$VC_{i,i} := VC_{i,i} + d$;
{ $VC_i = [c_1, \ldots, c_i + d, \ldots, c_n] \wedge e = W$ }
s_i:: π_k!(e, T_i);
{ $VC_i = [c_1, \ldots, c_i + d, \ldots, c_n] \wedge e = W$ }

R_j:: { $VC_j = \vec{L}$ }
r_j:: π_k?(x,temptime);
{ x = D ∧ temptime = $\vec{E}$ }
$VC_{j,j} := VC_{j,j} + d$;
VC_j := sup $(VC_j$, temptime);
{ x = $D \wedge VC_j = asynch_r(j, \vec{L}, \vec{E})$ }

$glue(s_i, r_k)$: $(after(r_k) \wedge T_k = \vec{F} \wedge$ indata = $Y \wedge$ intime = $\vec{Z})$
$\vdash ((after(s_i) \wedge T_i = \vec{F}) \vdash (e = Y \wedge VC_i = \vec{Z}))$
$glue(s_k, r_j)$: $(after(r_j) \wedge T_j = \vec{G} \wedge$ tempdata = $D \wedge$ temptime = $\vec{E})$
$\vdash ((after(s_k) \wedge T_k = \vec{G}) \vdash$ (outdata = $D \wedge$ outtime = $\vec{E}))$
$glue(s_i, r_j)$: $(after(r_j) \wedge$ tempdata = $D \wedge$ temptime = $\vec{E})$
$\vdash ((after(s_i) \wedge VC_i = \vec{E}) \vdash (e = D))$

Figure 5.10: Annotation of an implementation of a virtual circuit using CSP.

Chapter 6

Testing to Validate an Annotation

6.1 Introduction

Once we have written a causal annotation of a distributed program, and have used the annotation to map the program's state functions onto its specification, we need to validate our work by testing. As we explained in Section 2.2, our strategy is to log program state during test executions, and then to perform postmortem analysis of the trace records. The key to this strategy is that the assertions from our process proof outlines tell us what to trace, and that they and the glue predicates guide trace analysis.

Since the proof assertions are all recursive, we can record and analyze them effectively. Since they are chosen by the verifier to expose precisely those aspects of program state that affect specification satisfaction, we can keep the overhead of tracing as low as possible. And, since they reference only the state of one or of a pair of processes, we can check to see if they correctly describe the program without facing a combinatorial explosion of states.

We do not claim that this methodology answers every question about testing. We do not even consider the difficult questions of chosing test data or of drawing conclusions about test coverage. We assume that the tester will use standard black and white box testing methods for module and system testing like those suggested by Myers [Mye79].

What we do claim is that our technique clearly relates testing to verification and so to specification satisfaction. We do not need to consult an oracle to tell whether test output is satisfactory. The annotation describes program states and state transitions. We analyze traces to see if the program actually passed through those states during our test runs. If it did, our confidence in the validity of the annotation increases. If not, we know that the annotation is incorrect. The problem may be an error in the code, or it may be an error in the annotation. In either case, we can use the trace to help us find and correct the mistake, and then begin again the process of convincing ourselves that we have a valid proof and a correct implementation.

6.2 An Example: Validating our Causal Annotation of Minset

To illustrate, let us consider the annotation of the CSP program Minset given in Figure 6.1. We have added a statement number to the left of each line of code. We want to record one trace log for each process Least(i), and one for process B. Every time in the course of a test run that we reach an assertion, we need to record the value of each variable mentioned. We must also log the statement number and the value of the process vector clock. Even if the assertion does not reference the clock, we may need its value to validate glue assertions, and we can use it in other cases to identify particular instances of the assertion record.

Suppose that we use Minset to determine the minimum value in set $\mathcal{A} = \{1, 3, 6, 9\}$. We will need $N = 4$ processes, Least(1) through Least(4), to determine the minimum, plus one process B to receive and store it. Assume that Least(1) is initially assigned the value 1, Least(2) gets 3, Least(3) gets 6, and Least(4) gets 9.

In the first test run, Least(1) sends the tuple (1,1), representing its minimum and set size, to Least(2), and then terminates. Least(2) will set its minimum to 1 and its set size

```
      Least(i)::
1)    integer mymin, theirmin, mysize, theirsize;
2)    mymin, mysize := a_i, 1;
3)    {a_i = mymin ∧ mysize = 1}
4)    do
5)    []_{j=1:N∧i≠j} 0<mysize<N;
6)    {1≤j≤N ∧ a_i ≥mymin∧ [ (mysize=1 ∧ T_i = 0⃗) ∨ ((Nm:T_im≠0) = mysize) ] }
7)    α:   Least(j)!(mymin,mysize) →
8)               {a_i ≥mymin}
9)               mysize := 0
10)              {mysize=0}

11)   []_{k=1:N∧i≠k} 0<mysize<N;
12)   {a_i ≥mymin∧ [ (mysize=1 ∧ T_i = 0⃗) ∨ ((Nm:T_im≠0) = mysize)]}
13)   β:   Least(k)?(theirmin,theirsize) →
14)              {a_i ≥min(mymin,theirmin)∧ (Nm: T_im ≠ 0) = mysize+theirsize }
15)              mymin, mysize := min(mymin,theirmin), mysize + theirsize
16)              {a_i ≥mymin∧ (Nm: T_im ≠ 0) = mysize }
17)   od;
18)   {(mysize=0) ∨ ((Nm: T_im ≠ 0)=mysize=N))}
19)   if mysize=0 → skip {mysize=0}
20)   [] mysize=N →
21)         {(Nm: T_im ≠ 0)=mysize=N ∧ a_i ≥mymin}
22)                γ:    B!mymin
23)         {(Nm: T_im ≠ 0)=N+1 ∧ a_i ≥mymin}
24)   fi
      ─────────────────────────────────────────────
25)   B:: if []_{i=1:N}   δ:    Least(i)?m → skip fi
26)      {1≤i≤N ∧ m=W ∧ ((Nm: T_{(N+1)m} ≠ 0)=N+1)}
      ─────────────────────────────────────────────
```

Glue predicates:

1) $glue(\alpha_i, \beta_j) \equiv [\text{at}(\beta_j) \wedge T_j = \vec{X} \wedge T_{ji} = 0] \supset [\text{at}(\alpha_i) \supset (T_i \cdot \vec{X} = 0)]$
2) $glue(\alpha_l, \beta_j) \equiv [\text{after}(\beta_j) \wedge \min(\text{mymin}_j, \text{theirmin}_j) = Y \wedge T_j = \vec{Z}]$
   $\supset [\forall l\text{: } (\text{after}(\alpha_l) \wedge T_{ll} = \vec{Z}_l \neq 0) \supset \text{mymin}_l \geq Y)]$
3) $glue(\gamma_i, \delta) \equiv [\text{after}(\delta) \wedge m{=}W] \supset [\text{after}(\gamma_i) \supset \text{mymin}_i = W)]$

Figure 6.1: Annotation of Minset.

to 2. Meanwhile, Least(4) sends the tuple (9,1) to Least(3), which keeps its minimum at 6, but increases its set size to 2. Least(3) then transmits (6,2) to Least(2), which keeps 1 as its minimum, but increases its set size to 4. Least(2), because it now knows about 4 values, sends the value 1 to process B, and processing is finished.

We depict this test run graphically in Figure 6.2, and then show the trace logs for the 5 processes in Figure 6.3. Analysis of the individual process logs is easy. We simply check

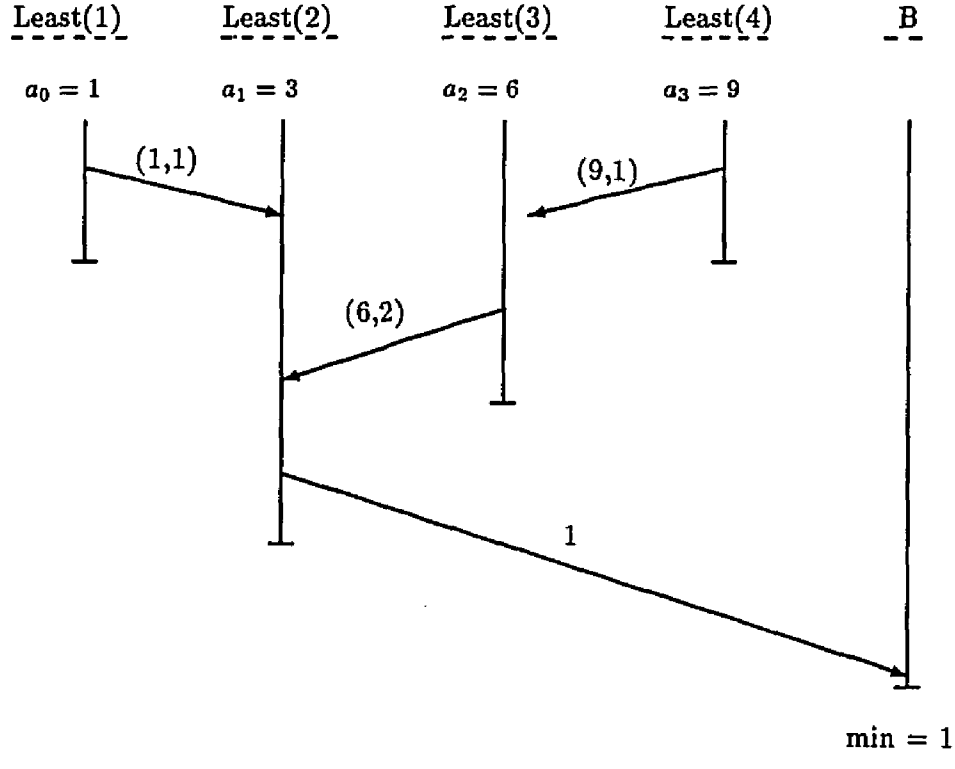

Figure 6.2: Minset test run.

that our local assertions accurately describe the variable and clock values for each process. For example, the assertion on line 10 in Figure 6.1 tells us that after a Least process has executed the assignment in line 9, the value of variable mysize should be 0. Looking in the trace logs of Least(1), Least(3) and Least(4) for the records associated with line 10, we see that the assertion holds in each case.

To test the validity of the glue predicates, we must look at pairs of trace records.

Process	Line	a_i	j	k	mymin	mysize	theirmin	theirsize	clock
Least(1)	3	1			1	1			[0,0,0,0,0]
	6	1	2		1	1			[0,0,0,0,0]
	8	1			1				[1,1,0,0,0]
	10					0			[1,1,0,0,0]
	18					0			[1,1,0,0,0]
	19					0			[1,1,0,0,0]
Least(2)	3	3			3	1			[0,0,0,0,0]
	12	3		1	3	1			[0,0,0,0,0]
	14	3			3	1	1	1	[1,1,0,0,0]
	16	3			1	2			[1,1,0,0,0]
	12	3		3	1	2			[1,1,0,0,0]
	14	3			1	2	6	2	[1,2,2,1,0]
	16	3			1	4			[1,2,2,1,0]
	18					4			[1,2,2,1,0]
	21	3			1	4			[1,2,2,1,0]
	23	3			1				[1,3,2,1,1]
Least(3)	3	6			6	1			[0,0,0,0,0]
	12	6		4	6	1			[0,0,0,0,0]
	14	6			6	1	9	1	[0,0,1,1,0]
	16	6			6	2			[0,0,1,1,0]
	6	6	2		6	2			[0,0,1,1,0]
	8	6			6				[1,2,2,1,0]
	10					0			[1,2,2,1,0]
	18					0			[1,2,2,1,0]
	19					0			[1,2,2,1,0]
Least(4)	3	9			9	1			[0,0,0,0,0]
	6	9	3		9	1			[0,0,0,0,0]
	8	9			9				[0,0,1,1,0]
	10					0			[0,0,1,1,0]
	18					0			[0,0,1,1,0]
	19					0			[0,0,1,1,0]

Process	Line	i	m	clock
B	26	2	1	[1,3,2,2,1]

Figure 6.3: Trace logs for Minset test run.

The first glue predicate, $glue(\alpha_i, \beta_j)$, relates the states of two least processes at the entry points of a communication in which one sends the other its minimum value. It states that before Least(i) transmits its minimum to Least(j), no process has communicated with both Least(i) and Least(j). A nonzero vector clock element k indicates that communication has occurred with process k. If the dot product of T_i and T_j is 0, then no element is nonzero in both vectors.

In the log of Least(2), there are two records for line 12, which corresponds to control point $at(\beta_2)$. In the first, $k = 1$, so we look for the matching send in line 6 (corresponding to $at(\alpha_1)$) of the log of Least(1). We see that $T_2 = T_1 = \vec{0}$, so the dot product is also 0. In Least(2)'s second record for line 12, $k = 3$, so we find the matching record for line 6 in Least(3)'s log. In this case, $T_2 = [1, 1, 0, 0, 0]$ and $T_3 = [0, 0, 1, 1, 0]$, so again the dot product is 0.

Similarly, in the log of Least(3), in the record for line 12, we see that $k = 4$ and $T_3 = \vec{0}$. In the matching record for line 6 in Least(4), $T_4 = \vec{0}$, and $T_3 \cdot T_4 = 0$.

The second glue predicate, $glue(\alpha_l, \beta_j)$ relates Least(j)'s state at the exit point of its receive, to the state at the exit point of the the tuple transmission of each process which has directly or indirectly communicated with Least(j). These are the processes for which T_j has nonzero elements. To check this predicate, we compare the records for line 14 in the receiver's log with those for line 8 in the appropriate senders' logs. We find that, after its first receive, Least(2) has $T_2 = [1, 1, 0, 0, 0]$, mymin = 3 and theirmin = 1, so min(mymin, theirmin) = 1. Only Least(1) is implicated by T_2, and in its line 8, mymin = 1, so the glue predicate holds. After its second receive by Least(2), $T_2 = [1, 2, 2, 1, 0]$, mymin = 1, theirmin = 6, and min(mymin, theirmin) = 1. This time we must check line 8 in the logs of all the other Least processes, but in each case the predicate holds. We also find that the predicate is true when we look at line 14 of Least(3)'s log and line 8 of Least(4)'s.

In the third glue predicate, $glue(\gamma_i, \delta)$, we assert that the minimum sent by a Least

process to process B and the value received by B are equal. To test this predicate, we must compare the line 23 record from Least(2)'s trace with that for line 26 from B's trace. Doing so shows that indeed mymin = m = 1. In this test run, then, our annotation has correctly described program state.

6.3 Another Example: Finding An Error in Minset

Of course, finding that our assertions correctly describe program state in one test run is not enough to convince us that we should have confidence in our annotation. As with any testing strategy, we must test until we are satisfied that we have exercised the code thoroughly. Even then, we will not be sure, in general, that our code is correct, but we can be more confident than if we had not taken an integrated approach to verification and testing.

In our case, like with all testing, we are really looking for errors, not for successful test runs. An errorless test cannot prove anything, but an error is a counterexample to the implication that we thought we had established from code to specification.

For example, suppose that in keying in the code for Minset, we mistakenly typed "B!mysize" rather than "B!mymin" in statement γ on line 23, transmitting the value of variable "mysize" rather than of "mymin". Typing errors like this can be hard to notice, because we often see in the code what we expect to find. If our test run had the same order of communication as in the previous example, our trace logs for the Least processes would be the same as in Figure 6.3, but the log for process B would be that shown in Figure 6.4. Though we might not notice the typographical error as we write the annotation, when we

Process	Line	i	m	clock
B	26	2	4	[1,3,2,2,1]

Figure 6.4: Log for process B in bad Minset test run.

check $glue(\gamma_i, \delta)$, we will see that $4 \neq 1$. This will tell us that something is wrong either

in the annotation or in the code, and scrutiny of the code will reveal the error. If the minimum of $\mathcal{A}$ were equal to N, we would not detect the error, but repeated testing with other values would soon expose it. We would then correct the error, and begin again the process of establishing confidence in our proof.

Other errors might be more subtle and therefore take more effort to expose, but the same principle applies. No test can demonstrate that code is correct, but careful testing of an annotation can make us more willing to believe that we have not made errors in our program proof.

6.4 Annotation Validation as a Practical Test Strategy

A significant concern for any practical testing methodology is the amount of overhead that it imposes on the system. When we trace program state, we use resources both of time and space, and we must attempt to keep the demands on each small enough to make tracing computationally feasible.

Trace logs can use large amounts of disk space. We try to minimize disk space requirements in two ways. First, we only record information considered significant by the verifier. We can safely ignore any aspect of process state not referenced in an assertion, since it does not affect the proof in which we are trying to build our confidence. This lets us filter out unwanted data, without losing the guarantee that we have maintained causal precedences.

Second, we only trace data relevant to the correctness of the module whose proof we are validating. Transition axiom specifications describe the states of modules. As we saw in Chapter 2, we specify a module in terms of its internal and interface state functions. Once we have established the correctness of a module, we can use it as a functional unit of a system without concerning ourselves with its internal state. We can define and prove a module correct, and then combine it with others to form more complex systems. At the

system level, we need only worry about module interaction.

We can take advantage of this modularity to reduce the amount of data we must record when we test. As we validate the annotation of a module, we need to record data about its internals as well as about its interactions with other modules. But, once we are confident that the module's implementation satisfies its specification, we no longer need to log its internal state transitions. Instead, we can focus on some other module, and record only those aspects of state which we have mapped onto the interface state functions of the specification. As we proceed up the hierarchy of specified modules, we can stop logging even the interface state functions of the lower level modules, which become internal to the functioning of the higher level modules of which they are a part.

By recording only significant data and by using the principle of modularity in testing, we also reduce the time overhead imposed by our testing model. Obviously, it takes time to log trace records. By filtering out extraneous data, we reduce the time it takes to write to the trace files. This reduces the *probe effect* [Gai86] of our tracing, that is, the distortion of an execution's timing dependencies due to the effects of testing.

The probe effect has the most serious consequences when caused by the imposition of additional message passing or synchronization, since these directly change the causal ordering of the program's events and states. We add neither messages nor synchronization. We change the ordering only to the extent that, one, the progress of processes is slowed by logging the important aspects of process state, and, two, that the transmission of messages is slowed by prefixing each message with a vector timestamp. We can limit the effect on message transmission by using our knowledge of the program and the system on which we implement it to bound the size of each timestamp element [LK90].

Time is also an important consideration when we analyze our trace logs, and here, too, our approach is practical. We saw in Chapters 2 and 3 that using causal rather than global thinking reduces the complexity of our analysis. Modularity also helps by cutting down the number of states that we must consider at any one time, while tracking

only meaningful data ensures that we waste no computational effort checking irrelevant conditions.

Though the overhead of tracing and trace analysis seems bearable, one might argue that the use of vector clocks makes our testing methodology feasible only for programs with a fixed number of processes, since we must fix the size of the clocks at compile time. We respond that we need only to be able to bound the number of processes, since a process starting up will have had no interactions with other processes and can initialize its clock with locally available information. For testing, we would not want to have an arbitrarily large number of processes, since we are not able to analyze an unlimited amount of information. Devising a test plan would almost certainly, therefore, include setting a limit on the number of processes in our tests.

We believe, then, that our strategy is a practical one for testing distributed programs. It lets the software developer integrate testing into a formal development methodology, without imposing unrealistic demands on the resources of either the tester or the computer system.

Chapter 7

Conclusions

In this thesis, we have presented an integrated approach to the specification, verification and testing of distributed programs. We reviewed Lamport's transition axiom method for the specification of safety and liveness properties, and showed how the "global" properties defined in transition axiom specifications can be interpreted as definitions of the causal relationships which must hold between local process states. We discussed the problems which arise from reasoning about the global state of a distributed program, and explained why reasoning about causal rather than concurrent relationships between process states yields a clearer picture of distributed processing.

We presented a system of axioms and rules to be used to prove the partial correctness of CSP programs. This proof system places strict restrictions on the types of assertions we may make when we derive a program annotation. It admits no global assertions. The annotation of a process may reference only locally defined variables, constant values and the process vector clock. *Glue* predicates relate pairs of process states at points of interprocess communication, using control predicates and vector clock values to identify the states. Additionally, no assertion may reference auxiliary variables; appropriate use of control predicates and vector clock values eliminates the need for them.

Our proof system emphasizes the importance of causality. We do not prove processes

correct in isolation. We do not derive "miraculous" postconditions for communication statements which must be justified after the fact by tying together isolated process annotations, nor do we postpone the derivation of useful postconditions until complete process proofs can be combined with a communication history invariant. We instead track causality as we write our process annotations. When we come to a send or receive statement, we consider all the statements which could communicate with it, and use the semantics of CSP message passing to derive the appropriate postcondition.

We showed that our CSP proof system is sound, that what we can prove in it is true. We also proved that, relative to some complete deductive system for the data types and operations of CSP, it is complete, that we can prove that any correct program is, in fact, correct. Though it seems reasonable to expect that the restrictions we place on our assertions would not affect the soundness of our proofs, it is less obvious that they would not make it less powerful than other, less restrictive, systems. Our proof of relative completeness ends this worry. It shows that, though we use only causal reasoning, make no global assertions and avoid auxiliary variables, we need to use only recursive assertions to prove that any program written in our fragment of CSP is partially correct. Our proof system is, therefore, as powerful as other proof systems for CSP.

We adapted and extended our work on the CSP proof system to develop proof systems for asynchronous communication. We provided axioms and proof rules for two asynchronous message–passing paradigms, unreliable datagrams and virtual circuits.

For each of these proof systems, our motivation was the same. We want to write program proofs which help a verifier show that the code satisfies its specification, while making only assertions which a program tester can use to define which aspects of process state should be traced during test runs, and checked during postmortem analysis. Neither global assertions nor auxiliary variables are readily traceable. We can trace the assertions that we may make, though, without having either to modify program code or to impose additional synchronization or message passing on the implementation. Avoiding assertions

about global state offers the added benefit of letting us avoid a combinatorial explosion of states during trace analysis, since we only have to consider single process states and pairs of process states, and never all possible interleavings or all consistent cuts.

One might ask why, if we go to all the trouble to verify a program's correctness, would we then want to test? We observe that a proof, like a program, is susceptible to error. By tracing and analyzing program state during testing, we can build our confidence that our proof is valid, that it correctly describes the states through which the program passes during execution. Too many programmers have an almost magical faith in verification. We want to promote the view that verification is but one step, albeit an important one, in the task of developing software that does the job that it is intended to do.

Our results suggest several avenues for future work.

- We have considered only proofs and tests of safety properties of distributed programs, but transition axiom specifications also define their liveness properties. We need to extend our methodology so that we can prove that an implementation exhibits the appropriate liveness properties, and then validate those liveness proofs during testing. This will let the verifier demonstrate total, as well as partial, correctness.

- We need also to extend our approach to other communication paradigms for distributed systems, such as remote procedure calls, flush channels [Ahu91] and the Ada rendezvous. It would also be interesting to determine if causal verification and testing can be made feasible for use with shared memory parallelism, where the grain of process interaction is often much finer than in distributed programming.

- The automation of trace analysis presents another fruitful direction for inquiry. Since both our local assertions and glue predicates are recursive, a program could use them to analyze trace records. One approach would be to dovetail process traces. The trace analyzer would start reading the trace log of one process, say π_i, checking assertions against log records, and continue until it reaches a blocking communication

command. Then, using the glue predicate for that command to identify the matching statement in some π_j, the analyzer would mark its place in π_i's trace, and analyze π_j's log until it reached the statement identified by the glue predicate or another blocking communication statement. Eventually, it would be able to use the glue predicate to see whether the states of π_i and π_j were correctly related when they communicated. In this way, the analyzer would work through all the process traces, printing an error message if an assertion did not correctly describe a process state.

- Though we developed our methodology for use with tracing and postmortem analysis, many programmers prefer to test and debug their programs interactively. With some modification, we could adapt our approach to allow proof validation through interactive testing. As with any interactive distributed debugger, we would have to avoid the ill consequences of the probe effect. We could use some kind of logical time rather than clock time to minimize its effects.

- Interactive testing would call for a means to display information about program state in such a way that the tester could grasp it quickly, yet accurately. Graphical user interfaces would make this more manageable, but in a system with more than a few processes, information overload would be a problem that would need to be resolved. Though we avoid reasoning about global state, we would still need to relate pairs of process states. When communication is synchronous, each communicating process blocks, so relating their states would not be difficult. When we analyze states related by indirect communication, or by asynchronous message-passing, though, the problem is harder. We might be able to overcome it by requiring that in such cases processes record their states for later analysis, or that they append extra state information to their outgoing messages for the use of the interactive debugger. With this information available, we could implement a distributed **assert** statement to notify the tester that the states are not in the correct relation.

- A hybrid approach that allows the tester to interact with the program, but that insulates testing from the constraints of real time, might prove the best solution. To implement this, we could record our trace logs during a test run, and then use them to support program replay. The tester could interact with, and possibly modify, the replayed program, rather than trying to interact nondestructively with an execution in some facsimile of real time. The traces would be available to the debugger for use whenever needed, so relating significant pairs of process states would present little problem. Trace analysis would still technically be postmortem, but the tester would have the sense of control and involvement that only interactive testing offers.

Our goal in all this research would be the same as the goal of this dissertation: to integrate verification and testing so as to make formal specification and proofs of correctness truly practical tools for software development.

Bibliography

[ABM79] K.R. Apt, J.A. Bergstra, and L.G.L.T Meertens. Recursive assertions are not enough – or are they? *Theoretical Computer Science*, 8:73–87, 1979.

[AFDR80] K.R. Apt, N. Francez, and W.P. De Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[Ahu91] M. Ahuja. An implementation of f-channels, a preferable alternative to fifo channels. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 180–187, Arlington, Texas, June 1991. IEEE.

[Apt81] K.R. Apt. Recursive assertions and parallel programs. *Acta Informatica*, 15:219–232, 1981.

[Apt83] K.R. Apt. Formal justification of a proof system for communicating sequential processes. *Journal of the ACM*, 30:197–216, 1983.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[Bat88] P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, Madison, Wisconsin, May, 1988.

[BBFM82] H.K. Berg, W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Prentice Hall, 1982.

[BdFG86] F. Baiardi, N. de Franesco, and G.Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, 1986.

[Boo87] G. Booch. *Software Engineering with ADA, 2nd Edition.* Benjamin/Cummings Publishing Co., 1987.

[Bur74] R.M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. IFIP, 1974.

[BW83] P. Bates and J. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255–264, 1983.

[CB89] B. Charron-Bost. Measures of parallelism of distributed computations. In B. Monien and R. Cori, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*. Springer-Verlag, 1989. Lecture Notes in Computer Science 349.

[CD88] E. Clarke and I. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988.

[Che89] W. H. Cheung. Process and event abstraction for debugging distributed programs. Technical Report CCNG T-189, Computer Communications Networks Group, University of Waterloo, 1989.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[Cli73] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[DDH72] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, 1972.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DLP79] R.A. DeMillo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[EH86] E. Emerson and J. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[ES88] E. Emerson and J. Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 123–172. Springer-Verlag, 1988.

[FC88] S. Feldman and C.Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, Madison, Wisconsin, May, 1988.

[Fid88] C.J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 130–140, Madison, Wisconsin, May, 1988.

[Flo67] R. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science XIX*, pages 19–32. American Mathematical Society, 1967.

[Gai86] J. Gait. A probe effect in concurrent programs. *Software — Practice and Experience*, 16(3):225–233, 1986.

[GG75] J.B. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, 1975.

[GMGK84] H. Garcia-Molina, F. Germano, and W. Kohler. Debugging a distributed computer system. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, 1984.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, 1981.

[GY76] S.L. Gerhart and L. Yelowitz. Observations of fallibility in applications of modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–206, 1976.

[HC68] G.E. Hughes and M.J. Cresswell. *An Introducion to Modal Logic.* Methuen and Co., 1968.

[HL74] C.A.R. Hoare and P.E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3:135–153, 1974.

[HMW89] D.P. Helmbold, C.E. McDowell, and J. Wang. Analyzing traces with anonymous synchronization. Technical Report UCSC-CRL-89-42, Board of Studies in Computer and Information Sciences, University of California at Santa Cruz, 1989.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa72] C.A.R. Hoare. Towards a theory of parallel programming. In *Operating System Techniques.* Academic Press, 1972.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Kel76] R.M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam80a] L. Lamport. The "Hoare logic" of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[Lam80b] L. Lamport. "Sometimes" is sometimes "not never". In *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.

[Lam83a] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[Lam83b] L. Lamport. What good is temporal logic? In *Information Processing 83 : Proceedings of the Ninth IFIP World Congress*, pages 657–668, 1983.

[Lam85] L. Lamport. Specification. In M. Paul and H.J. Siegert, editors, *Distributed Systems: Methods and Tools for Specification*, volume 190 of *Lecture Notes in Computer Science*, pages 270–285. Springer-Verlag, 1985.

[Lam87] L. Lamport. Win and sin: Predicate transformers for concurrency. Computer Science Technical Report Research Report 17, DEC System Research Center, 1987.

[Lam88a] D.A. Lamb. *Software Engineering: Planning for Change*. Prentice Hall, 1988.

[Lam88b] L. Lamport. Control predicates are better than dummy variables for reasoning about program control. *ACM Transactions on Programming Languages and Systems*, 10(2):267–281, 1988.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, 1989.

[LeL81] G. LeLann. Motivations, objectives and characterization of distributed systems. In B.W. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems: Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*, pages 1–9. Springer-Verlag, 1981.

[LG81] G.M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[LK90] W.S. Lloyd and J.P. Kearns. Bounding sequence numbers: A general approach. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 312–319, 1990.

[LMC87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[LO81] L. Lamport and Susan Owicki. Program logics and program verification. In D. Kozen, editor, *Logics of Programs Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 197–199. Springer-Verlag, 1981.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[MC81] J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, SE-7:417–426, 1981.

[MC88] B.P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California, June 1988. IEEE.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, 1980.

[Mil84] E.F. Miller. Software testing technology: An overview. In C. Vick and C. Ramamoorthy, editors, *Handbook of Software Engineering*, pages 359–379. Van Nostrand Reinhold, 1984.

[ML88] B. Miller and T. LeBlanc, editors. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging.* SIGPLAN/SIGOPS, Madison, Wisconsin, May, 1988. Workshop summary, p. xxi.

[MP81] Z. Manna and A. Pnueli. Temporal verification of concurrent programs. In *The Correctness Problem in Computer Science*, pages 308–312. Academic Press, 1981.

[MP88] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 201–284. Springer-Verlag, 1988.

[Mye79] J.G. Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[Owi75] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, Ithaca, N.Y., 1975.

[Owi76] S. Owicki. A consistent and complete deductive system for verification of parallel programs. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing*, pages 73–86, Hershey, Pa., 1976.

[Par87a] I. Parberry. *Parallel Complexity Theory.* Research Notes in Theoretical Computer Science. Pitman Publishing, London, 1987.

[Par87b] R. Parikh. Modal logic. In *Encyclopedia of Artificial Intelligence*, volume 1, pages 617–619. Wiley-Interscience, 1987.

[Pnu81] A. Pnueli. The temporal semantics of concurrent computation. In *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 1981.

[Pom84] G. Pomberger. *Software Engineering and Modula-2.* Prentice Hall International, 1984.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[Pri67] A. Prior. *Past, Present and Future.* Oxford University Press, 1967.

[Rei88] W. Reisig. Temporal logic and causality in concurrent systems. In F.H. Vogt, editor, *Concurrency 88: Proceedings of the International Conference on Concurrency*, pages 121–139, Hamburg, FRG, October 1988. Springer-Verlag.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, 1971.

[SK87] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice Hall International, 1987.

[Sou84] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, 1984.

[SS84] R.D. Schlichting and F.B. Schneider. Using message passing for distributed programming: Proof rules and disciplines. *ACM Transactions on Programming Languages and Systems*, 6(3):402–431, 1984.

[Wei71] G.M. Weinberg. *The Psychology of Computer Programming.* Van Nostrand Reinhold, 1971.

[Wit88] L. Wittie. Debugging distributed C programs by real time replay. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 57–67, Madison, Wisconsin, May, 1988.

[Woh85] C. Wohlin. Software testing and reliability for telecommunication systems. In D. Barnes and P.Brown, editors, *Software Engineering '86*, pages 27–42. Peter Pergrinus Ltd., 1985.

[YC79] E. Yourdan and L.L. Constantine. *Structured Design.* Prentice Hall, 1979.

[You80] E. Yourdan. *Structured Walkthroughs, 2nd Edition.* Prentice Hall, 1980.

VITA

William Samuel Lloyd was born in Richmond, Virginia, on November 2, 1952. A graduate of Thomas Jefferson High School in Richmond, he received his B.A. in Psychology from George Washington University, Washington, D.C., in 1976. After working in government and industry as a computer programmer and project leader, he received the M.S. in Mathematical Sciences/Computer Science from Virginia Commonwealth University, Richmond, in 1988. He expects to receive his doctorate in Computer Science from the College of William and Mary in August, 1991, and to join the faculty of West Georgia College, Carrollton, Georgia, in September, 1991.